UNIVERSITAT POLITÈCNICA DE CATALUNYA
DEPARTAMENT DE LLENGUATGES I SISTEMES INFORMÀTICS
MÀSTER EN COMPUTACIÓ

TESI DE MÀSTER

ACCELERACIÓ DE LA NAVEGACIÓ PER A ENTORNS VIRTUALS COMPLEXES

ESTUDIANT: JOAQUIM VILÀ BOU
DIRECTOR(S): ÀLVAR VINACUA PLA
PONENT:

DATA: 23/06/2008

ÍNDEX

1 INTRODUCCIÓ

En aquest capítol es posa al lector en el marc d'aquesta tesi: es fa una breu introducció al software de visualització del que es parteix, s'explica el projecte d'investigació en que s'emmarca la tesi, es justifica el treball realitzat, es fixen els objectius que es volen assolir i finalment, es mostra quina és l'estructura que segueix aquesta memòria.

1.1 El visualitzador Alice

Totes les tasques que es desenvolupen en el context d'aquest projecte es fonamenten en un software de visualització ja existent. Es tracta d'*Alice*, un paquet gràfic desenvolupat pel *MOVING Research Group* (grup de recerca en informàtica gràfica de la *UPC*) des de l'any 1993. L'aplicació, desenvolupada sobre la plataforma *OpenGL,* permet la visualització immersiva d'escenes poligonals d'elevada complexitat en temps real.

Alice està dissenyat per a que resulti còmode navegar a través dels entorns 3D que es carreguin. Ofereix múltiples mode de navegació, de manera que l'usuari pot escollir el que millor s'adapta a les seves necessitats. A més, dona suport a un gran nombre de dispositius d'entrada (*joystick*, *mouse, tracking systems*, ...), facilitant-ne encara més el control. Finalment, per tal d'aconseguir una navegació més realista, incorpora un sistema de detecció de col·lisions. Així, s'impedeix que l'observador pugui travessar objectes, parets, ...

Per altra banda, tot i que l'aplicació no està pensada per a l'edició, també es disposa de funcionalitats mínimes de selecció i manipulació. D'aquesta manera, es permeten fer modificacions als models, tot i que de forma molt limitada.

Una altra de les característiques d'aquest software és que funciona sobre múltiples plataformes. Això és així, tan des del punt de vista de l'arquitectura (*IRIX*, *PC-Windows*, *PC-Linux*), com des del punt de vista del dispositiu de visualització (sistemes *CAVE*, taules esteroscòpiques, cascs de realitat virtual o altres sistemes basats en estèreo actiu, passiu o mono). Així doncs, independentment de quin sigui l'entorn de treball utilitzat, l'aplicació hauria d'executar-se correctament.

Amb l'objectiu d'aconseguir una visualització més realista, el programa permet incloure textures en els polígons dels models. D'aquesta manera, s'aconsegueix una representació del detall molt bona en comparació amb el cost de renderitzar-la.

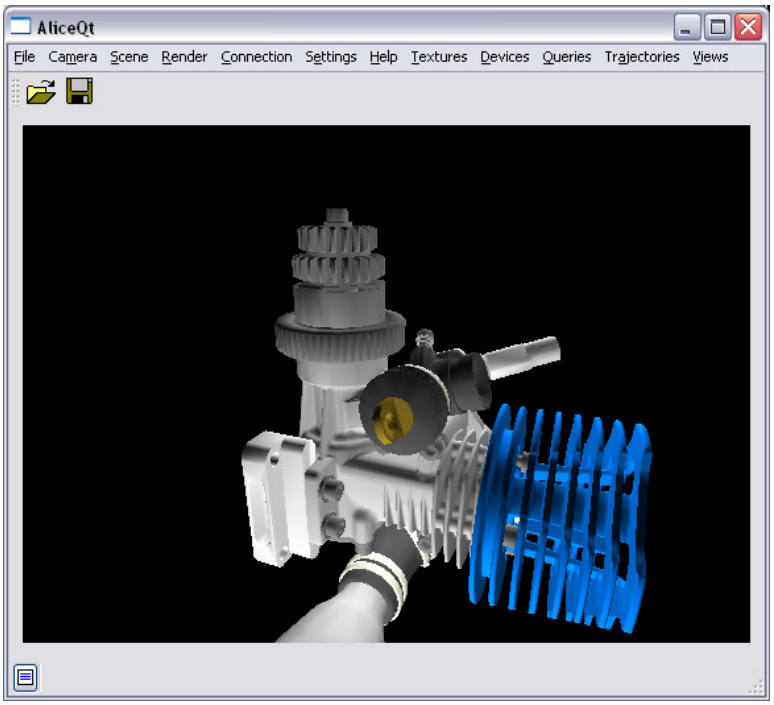


Fig. 1.1.1: *Screenshot del visualitzador Alice.*

Finalment, cal destacar que les primeres versions *d'Alice* tenien algunes funcionalitats addicionals. És el cas de la navegació mitjançant trajectòries predefinides o del control interactiu de la il·luminació. Malauradament, en el transcurs d'algunes evolucions de l'aplicació, aquestes comandes no han estat adaptades convenientment i s'han acabat deshabilitant.

1.2 El projecte BAIP 2020

El treball realitzat en aquesta tesi s'emmarca dins d'un projecte d'investigació finançat pel programa *CENIT* (*Consorcios Estratégicos Nacionales en Investigación Técnica*). S'anomena *BAIP 2020* (*Buque Autómata Inteligente Polivalente para la Pesca 2020*) i ha estat aprovat pel *CDTI* (*Centro para el Desarrollo Tecnológico Industrial*) dins la iniciativa *Ingenio 2010*, encaminada a col·laborar en grans projectes d'investigació industrial estratègica.

El projecte, amb una durada de 4 anys, disposa d'un pressupost de 39 milions d'euros. El seu objectiu és la investigació per al desenvolupament d'un nou concepte d'embarcació y de drassana, de manera que siguin més intel·ligents, polivalents i autònoms que els actuals. Amb aquest plantejament, es considera estratègicament innovador i per aquest motiu, s'ha convertit en el primer projecte *CENIT* del sector naval i marítim que aconsegueix l'aprovació del *CDTI*. Les principals línies d'investigació del projecte s'agrupen en els següents camps:

- Tecnologies de disseny i construcció naval
- Eficiència energètica i ús d'energies alternatives en les embarcacions
- Tecnologies i sistemes de pesca (pesca sostenible)
- Tecnologies oceanogràfiques per a la protecció i caracterització del medi marí
- Tecnologies en seguretat, confort i salut de la vida en el mar

Promogut per *Astilleros de Murueta* i *Sisteplant*, des del seu centre tecnològic de *Goldgym*, i amb el suport de *Innovamar*, el consorci creat per al desenvolupament del projecte agrupa a 21 empreses de 7 Comunitats Autònomes, així com 28 grups d'investigació d'Universitats i de Centres Tecnològics.

D'entre el grup d'empreses que integren en el consorci, en destaca la participació de *SENER*. Es tracta d'una enginyeria i consultoria que és la creadora de *FORAN*, un software de disseny i construcció naval líder en el mercat mundial. Així doncs, gràcies a l'àmplia experiència en la matèria, *SENER* és l'encarregada del projecte de treballar en el desenvolupament d'aplicacions que facilitin el disseny i la construcció d'embarcacions.

Fig. 1.2.1: *Logotips de SENER, FORAN i del grup de recerca MOVING de la UPC.*

Les tasques assignades a *SENER* està previst que les porti a terme en col·laboració amb el *MOVING Research Group* de la *UPC*. En aquest punt és on

apareix el vincle entre aquesta tesi i el projecte *BAIP 2020*. Donat que aquesta tesi es realitza en l'àmbit del *MOVING Group*, es pretén enfocar-la de manera que els resultats que se n'extreguin puguin utilitzar-se directament per aquest projecte *CENIT.*

1.3 Oportunitat de projecte

Tal i com s'ha comentat, un dels principals objectius d'*Alice* és la visualització en temps real. Això significa que l'aplicació ha de ser capaç de generar, com a mínim, entre 20 i 30 imatges per segon, encara que el model carregat sigui molt complex. En cas contrari, la navegació no serà suficientment fluïda i l'usuari apreciarà un retard no desitjable en la interacció. D'aquesta manera, esdevé imprescindible que el procés de *rendering* implementat sigui extremadament eficient.

Per tal d'aconseguir-ho, el paquet incorpora una sèrie d'optimitzacions que permeten reduir el temps necessari per a generar una imatge de l'escena. Concretament, les tècniques d'acceleració utilitzades són: el *back-face culling*, el *view frustum culling*, *l'occlusion culling* i la multiresolució.

Tot i això, cap d'aquestes millores es correspon amb els últims avanços en el món dels gràfics per computador, o fins i tot, de la informàtica en general. Així doncs, no s'estan tenint en compte aportacions importants, com és el cas de l'aparició de targetes gràfiques programables (*GPUs*) o bé la utilització de computadors amb múltiples processadors. Tampoc s'estan fent servir tècniques d'acceleració gràfica d'interès destacable, com són la utilització d'impostors, per a representar geometria distant, o bé la gestió de models en memòria externa (*out-of-core*), per a tractar models molt grans. Per tant, s'estan ometent contribucions molt potents, capaces d'accelerar notablement el procés de visualització de qualsevol aplicació gràfica.

Així doncs, la implantació d'algunes d'aquestes millores sobre *Alice* pot representar un increment considerable en el *framerate* resultant. Això suposa una excel·lent oportunitat per a desenvolupar una nova versió de l'eina. Cal estudiar quines optimitzacions són les més apropiades per al cas particular d'*Alice* (amb les quals es preveu obtenir un major increment de rendiment) i implementar-les.

D'aquesta manera, es disposaria d'un visualitzador actualitzat i amb unes prestacions superiors al disponible actualment (més eficiència i suport per a models de major complexitat). S'espera que així es pugui reactivar la seva utilització i a la vegada s'aturi la tendència al desús que ha sofert darrerament.

Per altra banda, un dels paquets de treball del projecte *BAIP 2020* consisteix en el desenvolupament d'una eina per a visualitzar els vaixells que s'estan dissenyant. Resulta indispensable que aquesta sigui capaç de suportar de forma interactiva models de grans embarcacions (complexitat molt elevada). Per tant, si es treballa per optimitzar *Alice* seguint la línia que s'ha plantejat, acabarà satisfent els requisits de l'aplicació que es necessita per al projecte i podrà ser utilitzada per a aquest propòsit. Això permet justificar que tot el treball que es preveu realitzar en el marc d'aquesta tesi té una utilitat garantida.

1.4 Estructura d'aquesta memòria

Un cop s'ha posat al lector en el context d'aquesta tesi, els següents apartats d'aquesta memòria expliquen tot el procés de treball que s'ha portat a terme.

En primer lloc, es defineixen els objectius que es volen assolir i es mostra la planificació de les tasques que es preveuen portar a terme al llarg del temps previst per a desenvolupar la tesi. A continuació, es presenta l'estudi de possibles optimitzacions a realitzar sobre *Alice*, vinculant-les amb l'estat de l'art de la temàtica a què fan referència. Tot seguit, es fa una valoració del guany que poden aportar en relació amb el temps necessari per a desenvolupar-les i es decideix quines s'implementaran.

Els següents punts contenen una descripció detallada del desenvolupament tècnic de les millores que s'han seleccionat. S'expliquen els principals canvis en el disseny de l'aplicació i es comenten els principals problemes detectats, així com la solució adoptada per a resoldre'ls. Seguidament, s'especifiquen els experiments realitzats per a mesurar l'increment de rendiment de l'aplicació i es presenten els resultats que s'han obtingut.

La part final d'aquesta memòria conté les possibles futures ampliacions d'aquest treball, una valoració de l'estimació realitzada en relació amb l'esforç real i les

conclusions que s'extreuen un cop acabat el projecte. A l'últim capítol s'hi troba la bibliografia, on es citen totes les fonts utilitzades.

2 OBJECTIUS

Una vegada s'ha demostrat que portar a terme aquest projecte és una bona inversió, cal clarificar quins són els objectius que es volen assolir una vegada s'hagi completat. Aquests es recullen en els següents dos punts:

- Fer un estudi que avaluï les diferents possibilitats per a accelerar el procés de visualització d'*Alice*, posant especial èmfasi en aquelles que pretenen optimitzar-lo aprofitant les prestacions del hardware actual.

- Implementar el subconjunt de les propostes anteriors que es consideri que pot produir un major increment en el rendiment de l'aplicació (subjecte al temps disponible), mesurar-ne els guanys i comprovar la seva correspondència amb els valors estimats a priori.

Donat que el temps disponible per al desenvolupament d'aquesta tesi és relativament limitat, no resulta possible determinar amb més precisió quines millores s'acabaran implementant. Un cop es finalitzi l'estudi, es farà una estimació del temps necessari per a la implantació de cadascuna i es seleccionaran les més apropiades en base a una planificació realista del temps del qual es disposa.

3 PLANIFICACIÓ I ESTIMACIÓ ECONOMICA

Una vegada es coneix quin és el treball que es vol portar a terme i s'han definit els objectius que es volen aconseguir amb aquest, cal determinar quines són les principals tasques que es preveuen fer. A més a més, també cal estimar el cost que es preveu que tindrà cadascuna d'elles per així poder distribuir correctament els recursos disponibles.

Així doncs, en primer lloc cal determinar de quins recursos es disposa per a poder distribuir-los posteriorment. El temps disponible per a la realització de tot el projecte és d'aproximadament 4,5 mesos (des de la segona meitat de febrer a final de juny) i s'estima una dedicació mitjana d'unes 30 hores setmanals. Si s'aproxima que un mes té 4 setmanes, aleshores:

4,5 mesos x 4 setmanes/mes x 30 h/setmana = **540 h disponibles**

D'aquesta manera, el nombre d'hores disponibles per a dur a terme tot el projecte és d'aproximadament 540.

Per altra banda, a partir dels objectius fixats es pot deduir en un primer nivell d'abstracció les tasques que caldrà fer. Es mostren a la taula 3.1.1, juntament amb el nombre d'hores que es preveu invertir en la realització de cadascuna, de manera que s'inverteixin totes les hores disponibles. S'ha de tenir present que la part de desenvolupament tècnic no es pot especificar amb un major grau de detall perquè no estarà ben definida fins que es completi l'estudi d'optimitzacions. Un cop finalitzat aquest, es farà una planificació amb una granularitat més fina del temps disponible per a la implementació de les millores que es seleccionin.

Un altre aspecte a tenir en compte és la distribució d'aquestes tasques al llarg de tot el període que ocupa el projecte. Cal organitzar en quin ordre es desenvoluparan per tal de realitzar-les de la manera més eficient possible. El diagrama de Gantt de la figura 3.1.2 mostra com es té previst fer-ho.

Nom de la tasca	Duració estimada
- Plantejament inicial	**60 h**
Definició del projecte	10 h
Selecció del visualitzador de partida	20 h
Preparació de l'entorn de treball i instal·lació de software	30 h
- Estudi de possibles millores per accelerar el rendiment d'*Alice*	**120 h**
GPUs i novetats d'OpenGL	15 h
Estructures de dades	15 h
Visibilitat	15 h
Multiresolució	15 h
Out-of-core	15 h
Multithreading	15 h
Documentació	30 h
- Desenvolupament tècnic de les optimitzacions seleccionades	**300 h**
Disseny i implementació	210 h
Disseny i execució d'experiments	30 h
Documentació	60 h
- Altres	**60 h**
Gestió del projecte (planificació, reunions, ...)	15 h
Confecció de la memòria	30 h
Presentació / lectura de la tesi	15 h
SUBTOTAL	**540 h**
CONTINGENCIA DE RISCS (10 %)	**54 h***
TOTAL	**594 h**

** En cas que sigui necessari, durant la fase final del projecte existeix la possibilitat d'augmentar el nombre d'hores disponibles per tal de cobrir possibles riscs.*

Taula 3.1.1: *Estimació d'hores necessàries per a desenvolupar cada tasca.*

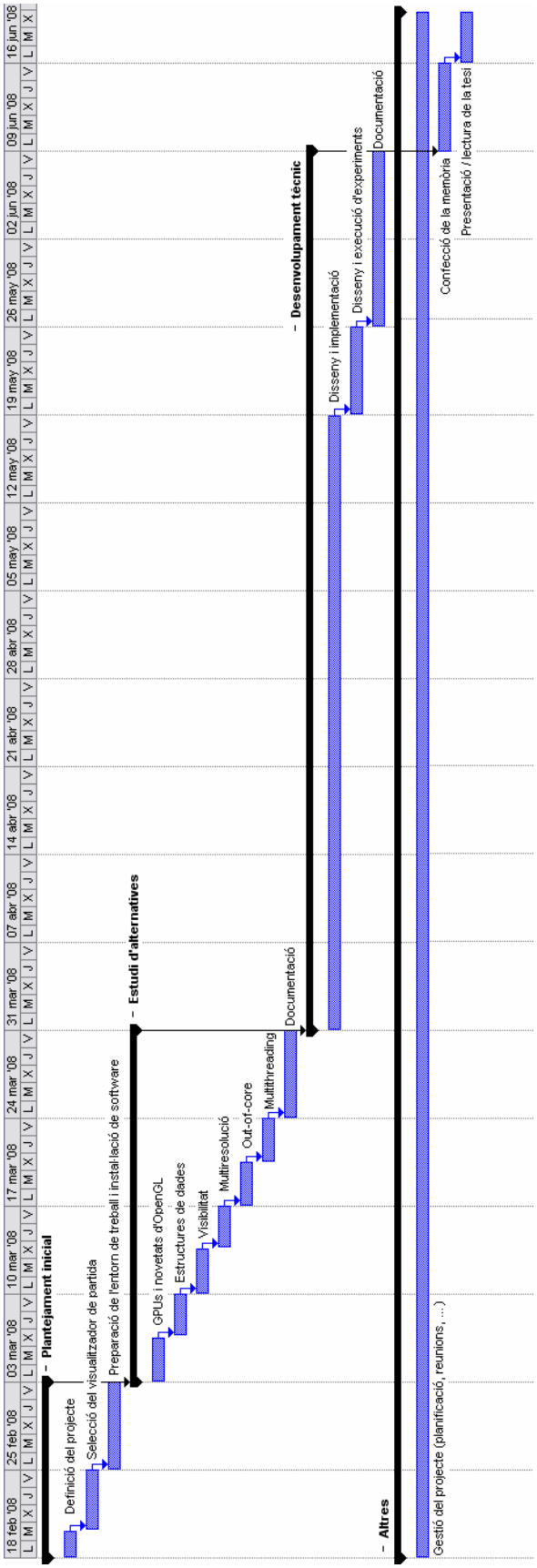


Fig. 3.1.2: *Diagrama de Gantt de les tasques planificades*

4 ANÀLISI D'ALTERNATIVES

En aquest capítol es planteja un ampli ventall de possibles millores per tal d'incrementar el rendiment d'*Alice*. Per cadascuna d'elles, es mostra l'estat actual de l'aplicació, es consideren diferents alternatives i s'analitza quina és la més apropiada, es comenten les principals discussions relatives als canvis que caldria realitzar, es fa una estimació del rendiment i finalment, s'indica si entra en conflicte o depèn d'alguna altra optimització. S'agrupen en els següents punts segons la temàtica que tracten: GPU i novetats d'OpenGL, estructures de dades, visibilitat, multiresolució, out-of-core i multithreading.

4.1 GPU i novetats d'OpenGL

Aquest punt considera una optimització relacionada amb la utilització de les noves capacitats de les targetes gràfiques programables (*GPUs*) i les extensions d'*OpenGL*. Es tracta de dos elements que han aparegut recentment i per això, no es tenen en compte en la última versió d'*Alice*. Així doncs, aquesta és una nova via d'explotació en la qual es poden desenvolupar millores amb un fort impacte.

4.1.1 Renderitzat eficient de dades geomètriques

Els models poligonals amb els quals treballa *Alice* consisteixen, en la seva essència, en una sèrie de dades geomètriques. Aquestes dades s'envien al hardware gràfic perquè les processi i en generi una visualització. Donat que la *CPU* i la targeta gràfica (*GPU*) són dos components separats, la transmissió d'informació entre aquests resulta molt costosa. D'aquesta manera, en aquest punt es planteja canviar el mecanisme d'enviament de dades geomètriques actual, per un altre que minimitzi la quantitat de dades transmeses.

4.1.1.1 Estat actual

Actualment, *Alice* envia a pintar les dades geomètriques utilitzant el *mode immediat* de *OpenGL.* Aquest mecanisme consisteix en enviar a la *GPU* totes les dades geomètriques del model carregat, a cada frame. Aquest esquema pot resultar vàlid en situacions com el modelat o l'animació, on es crea i es modifica

geometria amb molta freqüència. Tot i això, en el cas de la nostra aplicació, donat que es tracta d'un visualitzador de models, la informació pràcticament no canvia mai i per tant, gran part de la transferència de dades que es fa resulta innecessària.

Per altra banda, cal estudiar quina és l'estructura de la representació dels models 3D, per poder entendre el recorregut realitzat per l'algoritme de *rendering*. Tal i com es pot veure a la figura 4.1.1.1.1, *Alice* organitza els models mitjançant un *scene graph* o graf d'escena. Es tracta d'un arbre, on alguns dels seus nodes contenen una representació de la frontera d'un objecte polièdric (*B-Rep)*. Cada node *B-Rep* conté un conjunt de primitives geomètriques dels tipus que admet *OpenGL* (triangles, quadrilàters, polígons, anells de triangles, ...) i que en conjunt, formen un políedre.

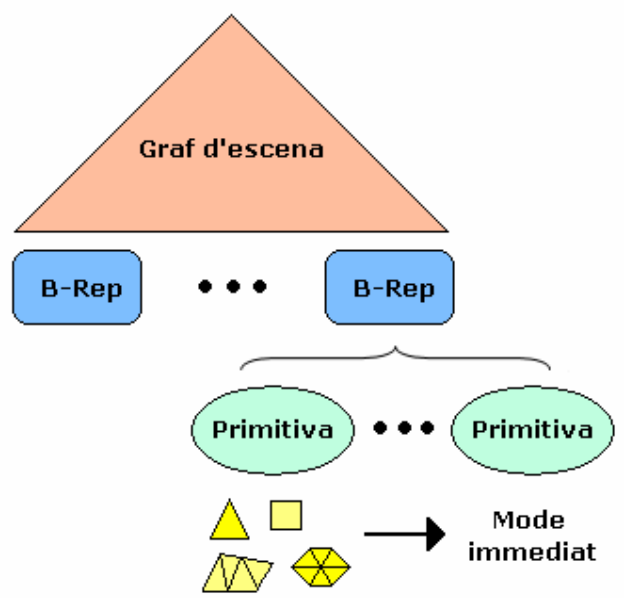


Fig. 4.1.1.1.1: *Esquema de l'estructura actual de la representació de models 3D.*

Així doncs, l'algoritme de *rendering* d'*Alice*, en termes generals, consisteix en un recorregut sobre tot l'*scene graph,* on per cada node *B-Rep*, s'envien a pintar en *mode immediat* les primitives que conté. Això resulta d'especial interès a l'hora de considerar alternatives al *mode immediat*, la majoria de les quals agrupa les dades a un nivell més gran que el de primitiva. D'aquesta manera, cal tenir molt en compte que amb alta probabilitat, qualsevol altra opció requerirà un canvi en la forma en com s'estructura l'escena.

4.1.1.2 Alternatives considerades

OpenGL ofereix tres alternatives al *mode immediat* per a realitzar el rendering de dades geomètriques de forma eficient [1]. Algunes d'elles passen per emmagatzemar geometria en la memòria de la targeta gràfica i així eviten la transmissió repetitiva de les mateixes dades. A continuació, es descriu en què consisteixen i es comenten els principals avantatges i inconvenients de cadascuna:

a. *Display Lists*: Consisteixen en una sèrie de comandes gràfiques, que s'enregistren a la memòria de la GPU i que tenen associat un identificador. D'aquesta manera, només amb que s'enviï aquest identificador n'hi ha prou per executar-les totes. Així, s'aconsegueix reduir notablement la utilització del bus de la targeta gràfica. Tot i això, hi ha un parell d'inconvenients que s'han de tenir presents. Un tracta sobre l'alt cost de canviar la geometria, ja que cada modificació implica regenerar la *display list*. Cal tenir molt en compte el cost associat a aquesta operació, sobretot si la freqüència de canvi és força alta. L'altre inconvenient es troba en que la *display list* es guarda per duplicat: en el client i en el servidor d'*OpenGL*. Això pot representar un veritable problema si la quantitat de memòria disponible està molt restringida.

b. *Vertex Arrays:* Són porcions de memòria que contenen conjunts de vèrtexs amb tota la seva informació associada (color, normal, coordenades de textura, ...). Agrupen i enregistren tota aquesta informació com un únic bloc, aconseguint transferències de dades molt eficients (es redueix el nombre de crides enviades). A més, es permet modificar les dades, tot i que s'ha de pagar el cost de validar-les a cada frame. També cal destacar que els *vertex arrays* no pateixen la limitació d'emmagatzemar dos cops la informació, tal i com passa amb les *display lists*.

c. *Vertex Buffer Objects (VBOs):* [2,3,4] Es tracta d'un tipus d'objecte molt similar als *vertex arrays*, però amb la particularitat que les dades s'emmagatzemen en memòria de la *GPU*, fent que el seu rendiment sigui notablement millor. Addicionalment, es creen *buffers* que indexen la informació dels vèrtexs i que agilitzen el renderitzat de primitives. Així doncs, els *VBOs* agrupen bona part dels avantatges dels altres modes i a la vegada n'eviten les limitacions. La informació es guarda de forma compacta i eficient, el *driver* de la targeta gràfica decideix quin és el millor lloc on

posar-la i es permet la seva modificació sense el cost d'haver-la de validar a cada *frame*.

Una vegada analitzades les tres possibilitats plantejades, els *Vertex Buffer Objects (VBOs)* semblen la opció més raonable degut a la gran quantitat d'avantatges que proporcionen. D'aquesta manera, en els següents punts es treballa sempre des de la perspectiva que aquesta és la opció que es vol desenvolupar.

4.1.1.3 Principals discussions

Per tal d'implantar sobre *Alice* un rendering que faci ús dels *VBOs*, cal discutir una sèrie de qüestions relatives als principals canvis que caldrà efectuar. Són les següents:

1. *Estructura de la representació del model*: Tal i com s'ha vist en el punt 4.1.1.1, la geometria del model carregat té una estructura que obliga fer-ne un renderitzat primitiva a primitiva. Aquesta estructura ve determinada pel format dels fitxers utilitzats per emmagatzemar les dades, el *P3D*. Així doncs, qualsevol canvi en l'organització de les dades del model passa per generar primer l'estructura actual a partir del fitxer *P3D* i després construir una nova estructura amb les característiques que es desitgin.

   Per altra banda, els *VBOs* obtenen el seu màxim rendiment quan treballen amb conjunts de dades més grans que simples primitives *OpenGL*, com ara malles de triangles. Crear un *VBO* a partir d'agrupar primitives no és una opció vàlida, ja que no es té en compte la relació entre aquestes i per exemple, els vèrtexs de primitives contigües es guardarien repetits o no s'assegura considerar la localitat espacial. A més, tampoc està garantit que totes les primitives que es prenguin siguin del mateix tipus i aquest és un requisit indispensable per als *VBOs*.

   Per tant, resulta necessari canviar l'estructura actual per una que agrupi la informació a nivell de malla. D'aquesta manera, es podria passar a renderitzar tota una malla de cop, utilitzant les comandes que ofereixen els *VBOs*. Una bona manera d'agrupar primitives sobre *Alice* és a nivell de políedre (Fig. 4.1.1.3.1): les primitives que els defineixen comparteixen molts vèrtexs i la localitat espacial està assegurada. Addicionalment, donat

que l'*scene graph* té a les seves fulles nodes de tipus *B-Rep* (representació de la frontera d'un políedre), els canvis estructurals que cal fer no són excessius.

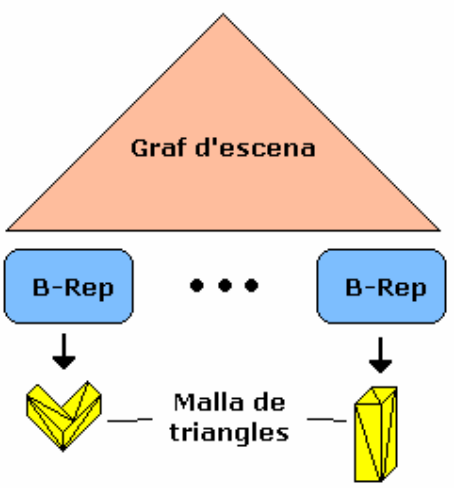


Fig. 4.1.1.3.1: *Esquema de l'estructura de models 3D basada en malles de triangles.*

El tipus de primitiva ideal per a les malles a formar és el triangle. *OpenGL* està optimitzat per a treballar-hi, i de fet, quan se li envien altres elements, internament els triangula. Tot i això, els models d'*Alice* poden estar composats per una mescla dels diferents tipus que admet *OpenGL*. A efectes d'utilitzar *VBOs*, això resulta un problema. Es requereix que cada paquet que s'enviï a pintar, tingui totes les primitives del mateix tipus. D'aquesta manera, la opció més raonable consisteix en triangular els quadrilàters o polígons existents (Fig. 4.1.1.3.2), per tal que les malles obtingudes estiguin formades exclusivament per triangles.

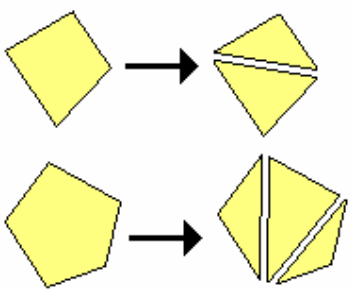


Fig. 4.1.1.3.2: *Triangulació de quadrilàters o polígons.*

Finalment, cal destacar que *Alice*, tot i ser un visualitzador de models, disposa de mecanismes selecció i edició de l'escena. Aquestes funcionalitats estan basades en l'estructura actual, on les primitives s'emmagatzemen per separat. Això significa que si es fan els canvis proposats, caldrà adaptar aquests mecanismes convenientment per tal que segueixin funcionant.

2. *Mida:* La mida en la qual els *VBOs* obtenen el seu màxim rendiment coincideix amb el tamany de la *pre-T&L vertex cache* [5]. Això, com és natural, és una característica que depèn de la targeta gràfica que s'utilitzi i que a més a més, es mou en un rang de valors força ampli. D'aquesta manera, s'hauria d'escollir un valor situat en una posició intermèdia, aproximadament uns 50.000 vèrtexs. Així, es podria tenir en compte un gran ventall de targetes, però sense haver de perdre's gran part dels avantatges dels *VBOs*. En qualsevol cas, caldria realitzar alguna prova en diferents plaques per tal de concretar amb més exactitud aquest valor.

   Tot i disposar d'un valor òptim, aquest només dona una orientació sobre el tamany definitiu dels *VBOs* que finalment es creïn. Així doncs, s'utilitzarà per determinar la granularitat amb què s'han d'agrupar els elements de l'escena per formar cadascun dels *VBOs* desitjats.

   Una primera aproximació consistiria en crear un *VBO* per cada políedre. En aquest cas però, el nombre de vèrtexs d'un políedre probablement serà molt inferior a 50.000. Això significa que caldrà agrupar-ne uns quants a cada *VBO*. A més, tal i com s'ha comentat, interessa fer aquesta agrupació segons un criteri de localitat espacial. Per tant, una estructura jeràrquica de divisió de l'espai esdevé necessària. Donat que actualment *Alice* no en té cap, en el punt 4.2.2 es proposa incorporar-ne una.

   D'aquesta manera, per determinar quins *VBOs* s'han de crear i quin contingut han de tenir es fa un recorregut sobre l'estructura de divisió de l'espai disponible. Segons la forma d'arbre d'aquest tipus d'estructures, es comença per l'arrel i es va baixant cap a les fulles, fins que es troba un node que conté menys vèrtexs que el valor de referència. Aleshores es crea un *VBO* i s'atura el recorregut per aquella branca.

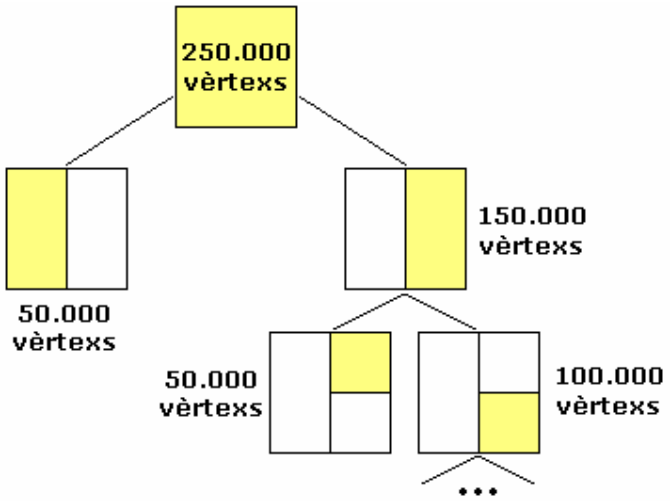


Fig. 4.1.1.3.3: *Exemple de càlcul dels VBOs a construir utilitzant una estructura jeràrquica de divisió de l'espai.*

3. *Política de gestió*: Cal tenir en compte que l'aplicació ha de ser capaç de visualitzar models de dimensions arbitràries i que la memòria de la targeta gràfica és limitada. Això pot provocar que no tots els *VBOs* tinguin cabuda dins la *GPU* i que alguns passin a emmagatzemar-se a la memòria principal. D'aquesta manera, resulta necessari aplicar algun tipus de política per gestionar quins VBOs es guarden a la targeta gràfica i quins no. Davant d'aquest punt, existeixen dues alternatives possibles:

    a. *Utilitzar la política que proporciona el driver*: Consisteix en crear tots els *VBOs* al carregar el model (incrementa el cost de la lectura) i deixar al *driver* de la targeta gràfica que gestioni en quin espai de memòria es guarda cada *VBO* en cada moment. Aquesta opció resulta molt fàcil de portar a terme (ja es troba implementada), però està preparada per a obtenir bons resultats en el cas general i per tant, podria ser que no fos la òptima per el tipus de navegació que es realitza. En qualsevol cas, no hi ha documentació disponible per poder verificar quin és el seu comportament. L'únic que es pot configurar són una sèrie de *hints* que permeten indicar al *driver* quin és l'ús que es farà de la informació emmagatzemada a cada *VBO*. Addicionalment, es pot tenir un mínim control de quins *buffers* han d'estar a la placa gràfica a partir d'enviar a pintar un únic triangle de cadascun. Així es pot donar una petita indicació al *driver* sobre quina informació és la que interessa tenir més accessible.

b. *Implementar una nova política basada en la navegació:* Es tracta de gestionar els *VBOs* de manera que només hi hagin aquells que s'estiguin visualitzant en cada moment o que es preveu que ho faran pròximament. Així, es va actualitzant el contingut dels *VBOs* segons evoluciona la navegació. Aquest mètode té l'avantatge que tot el que s'utilitzi es trobarà a la *GPU*, a part de tenir un major control de la forma en com es gestiona la informació (no depenent del *driver*). Tot i això, cal tenir en compte que desenvolupar un algoritme que efectuï aquesta gestió pot resultar una tasca relativament complexa. A més, es té el cost addicional de la transferència de dades cada cop que es decideixi el contingut dels *VBOs*.

Tot i els prometedors avantatges d'implementar una política pròpia, la complexitat que comporta fa que es desestimi aquesta alternativa. Es confia que el comportament del *driver* sigui acceptable i es considera que el mínim control que proporciona és suficient per als propòsits de l'aplicació.

4.1.1.4 Pronòstic de rendiment

Després d'un primer anàlisi del rendiment de l'aplicació utilitzant un eina de *profiling*, s'ha pogut veure que l'aplicació es passa la major part del temps enviant crides de pintat a la *GPU*. Per aquest motiu, és molt probable que un dels colls d'ampolla de l'aplicació es trobi en la comunicació entre *CPU* i *GPU* a través del bus *AGP*. Així doncs, donat que aquesta optimització es centra precisament en agilitzar aquest punt, és d'esperar que la implementació de *VBOs* aconsegueixi millorar notablement el rendiment de l'aplicació.

4.1.1.5 Conflictes i dependències

El desenvolupament d'aquesta optimització pot entrar amb conflicte o dependre d'alguna de les altres millores que es proposen. Són les següents:

1. *Nou format dels fitxers P3D (punt 4.2.1):* La utilització de *VBOs* requereix d'una estructura de dades diferent a l'emmagatzemada en els fitxers *P3D*. Si es modifica el format d'aquests fitxers, les dades dels models quedaran enregistrades segons la nova organització, i per tant, es podrà construir l'estructura necessària de forma directa. Altrament, caldrà carregar primer

l'estructura que s'utilitza actualment i després executar un procés que la converteixi al format necessari per als *VBOs*. En aquest segon cas, com és evident, es tindrà una penalització considerable en el temps de carrega del model.

2. *Estructura de divisió de l'espai (punt 4.2.2)*: Per tal de distribuir els vèrtexs del model entre els diferents *VBOs*, es proposa utilitzar una estructura jeràrquica de divisió de l'espai. Donat que *Alice* actualment no en disposa de cap, la implementació d'una esdevé necessària.

3. *Out-of-core (punt 4.5):* Si s'implanta alguna de les tècniques de gestió de memòria externa, per a poder visualitzar models de tamany arbitrari, caldrà tenir-ho en compte a l'hora de gestionar els *VBO.* En aquest cas, resulta necessari adaptar la política que els distribueix entre els diferents tipus de memòries per a que, a més de la memòria de la targeta gràfica i la memòria principal, consideri també el disc.

4.2 Estructures de dades

La manera com s'organitza la informació dels models que es visualitzen és un element clau pel rendiment d'*Alice*. Una bona estructura ha de permetre accedir de forma ràpida a la informació desitjada, basant-se només en uns simples criteris. De fet, moltes de les optimitzacions presentades en aquest treball tenen com a requisit un tipus d'estructura determinat. Així doncs, en aquest apartat es proposen diverses modificacions a l'estructura actual de l'aplicació, se'n valora la seva viabilitat i es relacionen amb aquelles millores que possibiliten.

4.2.1 Nou format dels fitxers P3D

El format dels fitxers que contenen els models d'*Alice* (*.P3D*) condiciona la informació que s'enregistra de forma persistent. Això significa que qualsevol canvi en l'estructura de dades, o qualsevol inclusió d'informació redundant per optimitzar els còmputs, s'haurà de calcular cada vegada que es carregui un model. Tenint en compte que la majoria de millores que es proposen utilitzen informació addicional, aquest tipus de modificacions esdevenen imprescindibles. En funció de la magnitud dels canvis, aquest fet pot tenir repercussions negatives importants en el procés de lectura de models.

Per tal d'evitar aquesta situació, en aquest punt es planteja utilitzar un nou format pels fitxers *P3D.* Així, es preveu poder realitzar tot aquest tipus de càlculs en temps de preprocès. Tot seguit, s'especifica quina és l'estructura actual d'aquests fitxers i com s'han d'adaptar per poder afrontar correctament les diferents optimitzacions proposades.

4.2.1.1 Estat actual

L'estructura de dades dels models que carrega *Alice* està basada en un *scene graph* [6]. Bàsicament, es tracta d'un arbre de nodes que es va construint a mesura que es llegeix el fitxer *P3D* d'entrada, utilitzant una estratègia *top-down.* D'aquesta manera, s'inicia la lectura pel node arrel i es prossegueix per cadascun dels fills, de forma recursiva. Aquest arbre s'anomena jerarquia de visualització i és el que utilitza el procés de rendering per a generar la corresponent imatge de l'escena.

Addicionalment, poden haver-hi altres arbres paral·lels, anomenats jerarquies de selecció, que permeten agrupar els nodes contenen geometria segons diferents criteris. Aquests nous arbres es fan servir per a gestionar les funcionalitats de selecció i manipulació d'objectes.

Pel que fa als nodes, es classifiquen en diferents tipus en funció del concepte que representen. Cadascun guarda les seves pròpies dades i té un comportament particular. Son els següents:

- *B-Rep*: Situat a les fulles de *l'scene graph,* aquest node emmagatzema una representació de la frontera d'un objecte 3D, la qual consisteix en una llista de primitives *OpenGL* (punts, triangles, strips, ...). Totes aquestes primitives pertanyen a una mateixa jerarquia de classes, de manera que resulta molt simple incorporar nous tipus de primitives no suportades per *OpenGL.* Tot i això, no es guarda cap tipus d'informació relativa a la connectivitat entre aquestes. Tampoc es fa cap mena d'ordenació basada en la distribució espaial dels objectes.

- *LOD*: Indica que un objecte disposa de múltiples representacions, en diferents nivells de detall, on cadascun dels fills del node coincideix amb una d'aquestes representacions. Durant el *rendering*, s'escull el nivell de detall més adient, i només s'envia a la targeta gràfica la geometria corresponent al fill seleccionat.

- *Light*: Son la representació d'un focus de llum, actualment de tipus puntual. Aquest node pot situar-se en qualsevol lloc de l'arbre, però el focus que defineix només il·lumina la geometria situada en aquelles branques que parteixen d'ell.

- *Text*: Permet la visualització d'un text dins la finestra d'*OpenGL*. Aquest pot localitzar-se sobre el pla de la pantalla o en un punt 3D de l'escena.

- *Picture2D*: Conté una imatge 2D que es visualitza sobre el pla de la pantalla i que pot utilitzar-se per a dibuixar, per exemple, un plànol de l'escena per la que s'està navegant.

- *CameraRep*: Representació gràfica de l'observador. Consisteix en un avatar situat a la posició on es troba aquest i que permet identificar on es troben altres participants quan es treballa amb entorns col·laboratius.

- *OccluderRep*: S'associa a algun dels oclusors que l'usuari pot definir de forma manual i que s'utilitzen per accelerar el procés de visualització. Es guarda una representació d'aquest objecte per poder seleccionar-lo i canviar-ne les propietats.

- *SelHierarchyNode*: Permet agrupar els objectes de l'escena (nodes *B-Rep)* per formar les jerarquies de selecció. Com que només tenen aquest propòsit, no enregistren informació addicional.

Per tal d'il·lustrar gràficament aquesta estructura, en l'exemple de la figura 4.2.1.1.1 es pot veure una instanciació del graf d'escena d'*Alice*. S'hi poden apreciar de forma diferenciada l'existència de dues jerarquies: la de visualització (arbre de l'esquerra) i una de selecció (arbre de la dreta).

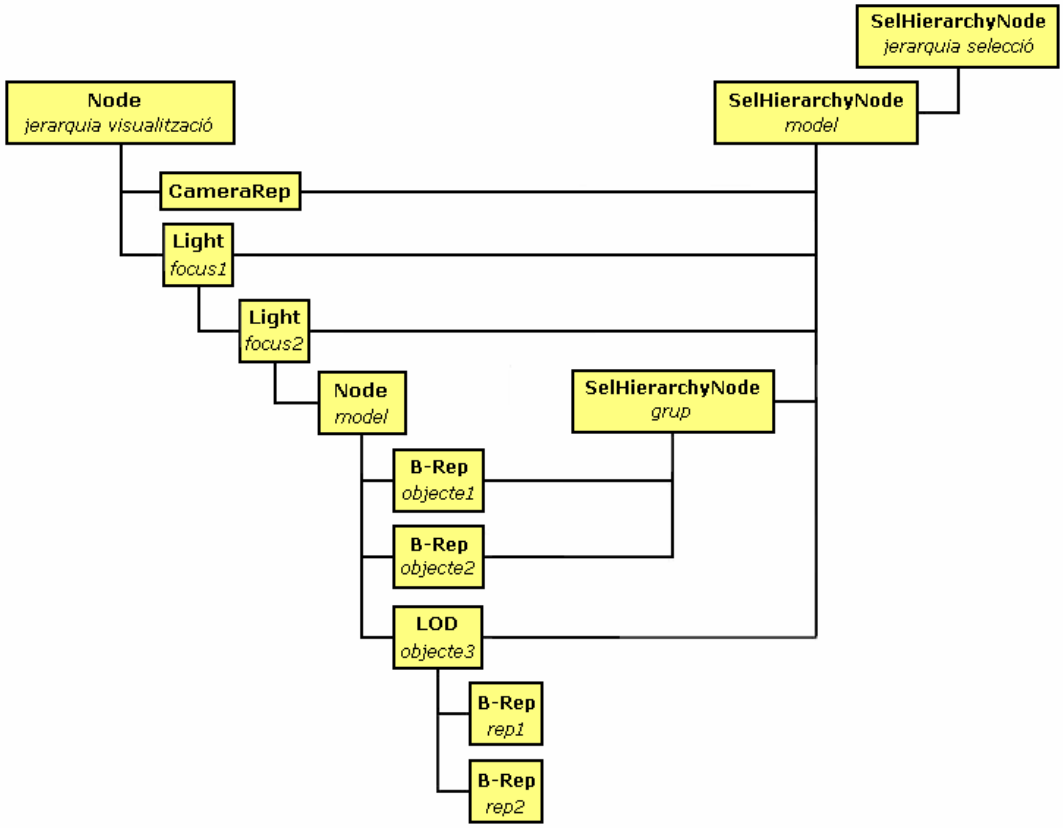


Fig. 4.2.1.1.1: *Exemple d'instanciació del graf d'escena que implementa Alice.*

Cal remarcar que el format actual dels fitxers *P3D* únicament conté informació per a carregar l'*scene graph* que s'ha descrit. Qualsevol altra estructura auxiliar que es pretengui fer servir, queda fora de l'àmbit de dades que es guarden de forma persistent.

4.2.1.2 Alternatives considerades

A partir del conjunt d'optimitzacions que es plantegen en aquest treball, s'extreu un llistat que conté la nova informació que es calcula en cadascuna de les millores i que s'utilitza per a accelerar el procés de visualització. És el següent:

1. *Nova representació de la geometria (punt 4.1.1)*: La implantació de *VBOs* necessita que la informació geomètrica del model estigui representada a partir de malles de triangles i no segons sopes de primitives. Aquesta nova estructuració de les dades es preveu que es generi un cop s'ha carregui el graf d'escena actual. Per aquest motiu, es considera interessant fer persistent aquesta nova organització de la geometria.

2. *Estructura de divisió de l'espai (punt 4.2.2):* Per a la implantació d'algunes de les optimitzacions proposades, en particular aquelles que es basen en un *rendering* dependent de la vista, fa falta una estructura de dades que tingui en compte la distribució espaial dels objectes de l'escena. Per poder evitar haver de construir-la a cada lectura que es faci, es planteja tenir-la enregistrada en el propi fitxer *P3D.*

3. *Jerarquia multiresolució (punt 4.4.1)*: En aquest punt es planteja incorporar representacions simplificades en els nodes de la jerarquia de divisió de l'espai que es construeixi. Aquestes representacions de menor nivell de detall són un bon candidat per a incorporar dins els arxius de models i poder evitar-ne així la repetida creació.

4. *Impostors (punt 4.4.2)*: Es tracta d'una situació molt similar al cas anterior, però en aquest cas la informació que es vol evitar recalcular són les imatges utilitzades per texturar els polígons impostors. A més, en aquesta ocasió, el més natural sembla emmagatzemar les imatges com a fitxers independents i incloure-hi en els *P3D* una referència (tal i com es fa en el cas de les textures).

Donat que tots els punts de la llista anterior fan referència a informació que necessita càlculs de certa envergadura per a obtenir-se, s'opta per incloure-la tota en una nova versió dels arxius *P3D* (sempre hi quan es desenvolupin les optimitzacions a les que fan referència). S'ha de destacar que per dur a terme aquesta modificació, només cal adaptar convenientment les rutines de salvat i de lectura de models. La part relativa als càlculs (que és on es troba la complexitat del problema) ja s'haurà fet durant de la implementació de cadascuna de les millores afectades. Així doncs, en aquest punt únicament es planteja el fet de moure en temps de preprocès tot aquest tipus de còmputs.

4.2.1.3 Principals discussions

El desenvolupament del canvi en el format dels fitxers *P3D* té alguns punts crítics que cal discutir. Tot seguit, es descriuen les principals qüestions conflictives i es valoren les diferents possibilitats per a resoldre-les:

1. *Conservació de la informació prèvia:* Part de la nova informació que es vol enregistrar és redundant respecte a la ja existent. És el cas de les noves estructures de dades auxiliars que complementen *l'scene graph* que hi ha actualment a *Alice*. Davant d'aquesta situació, es planteja el dubte sobre si mantenir o no la dualitat d'aquestes dades. L'anàlisi de les dues opcions és el següent:

    a. *Mantenir informació redundant*: No cal preocupar-se de detectar en quins punts es fa servir el graf d'escena, ja que aquesta informació no deixarà d'existir. Tot i això, si que cal tenir present en quins llocs es modifica (comandes de manipulació), ja que al tenir informació redundant, cal procurar mantenir la consistència de les dades. Finalment, s'ha de considerar l'elevat consum de memòria que la dualitat d'estructures requereix. Aquest pot ser un aspecte molt rellevant en models molt grans, on els recursos disponibles són molt més limitats.

    b. *Eliminar la informació prèvia*: Aquesta alternativa requereix modificar alguns dels mecanismes ja existents a *Alice*, com són la selecció o la manipulació, els quals es basen en la informació suprimida. De totes maneres, la visualització, que és la part més complexa que fa servir l'*scene graph*, passarà a utilitzar les noves estructures. A més, la

gestió de la informació esdevindrà més simple donat que no s'ha de treballar de forma paral·lela amb dues estructures. Pel que fa a memòria, la quantitat necessària serà molt menor, cosa que suposa un avantatge important.

Es decideix adoptar l'alternativa de mantenir la informació redundant. Encara que es desenvolupi un procés de visualització totalment nou, *Alice* està molt lligat al graf d'escena i eliminar-lo per complet suposa un volum de feina molt elevat.

2. <u>*Ubicació del procés de conversió:*</u> Per poder utilitzar els models ja existents en la nova versió *d'Alice*, esdevé imprescindible executar un procés de conversió per poder adaptar-los al nou format. Aquest es pot situar en dos possibles punts:

    a. *En el procés de càrrega del model:* Consisteix en comprovar si el model que es carrega es correspon amb l'última versió del format *P3D* i en cas negatiu, realitzar la conversió (amb tots els càlculs que implica). Tot seguit, es procediria al salvat del model convertit. D'aquesta manera, únicament es penalitza la primera lectura de cada model i evita a l'usuari haver de preocupar-se per un canvi en les versions. Tot i això, aquesta proposta complica encara més la lectura de models degut a un control de versions del format *P3D* més complex.

    b. *En un conversor independent*: Es tracta d'utilitzar una aplicació independent d'*Alice* que s'encarregui de fer la conversió al nou format. Té com a principal avantatge que no s'augmenta la complexitat del procés de càrrega/salvat de models, ja que se l'allibera de la gestió de múltiples versions. Com a contrapartida, hi ha el fet que es força a l'usuari a que sigui ell mateix qui inici el procés de conversió, amb totes les incomoditats que això li pugui comportar.

Donat que cap de les dues opcions sembla clarament millor que l'altre, es decideix conservar la política adoptada fins ara i utilitzar un conversor independent. Per tant, s'allibera el codi d'*Alice* de tot aquest tipus de càlculs.

4.2.1.4 Pronòstic de rendiment

Les modificacions proposades en aquest punt estan centrades en millorar el rendiment del procés de càrrega de models. De fet, es preveu que aquest es vegi incrementat de manera preocupant pel desenvolupament d'algunes de les altres millores plantejades en aquest treball. Això pot suposar que l'usuari hagi d'esperar una quantitat de temps poc raonable per a llegir un model. En aquesta situació, resultaria d'especial interès poder moure els nous còmputs en temps de preprocès. D'aquesta manera, s'espera que el temps necessari per a poder carregar un model sigui de l'ordre del que fa falta actualment. Tot i això, com que encara no s'ha implementat cap de les noves optimitzacions, no resulta possible quantificar l'increment del temps de càrrega, i per tant, es fa difícil determinar si els canvis que aquí es plantejats compensen.

4.2.1.5 Conflictes i interferències

El desenvolupament d'aquesta proposta té una relació de dependència directa amb les altres millores que es tracten en aquest treball. Així doncs, es necessita haver desenvolupat prèviament aquelles optimitzacions de les quals s'enregistra la nova informació (descrites a la llista del punt 4.2.1.2). Altrament, no es disposaria dels procediments de càlcul que generen la nova informació que es vol incloure en els arxius.

4.2.2 Estructura de divisió de l'espai

La informació geomètrica que actualment utilitza *Alice*, tot i estar estructurada mitjançant un *scene graph,* no té en compte la seva distribució espacial. Tot i això, conèixer quina geometria hi ha a cada regió de l'escena pot resultar d'especial interès. Per exemple, es pot fer servir per detectar eficientment quins objectes es troben més a prop de l'observador i utilitzar aquestes dades per aplicar després, tècniques que agilitzin la navegació. Les estructures jeràrquiques de divisió de l'espai gestionen tot aquesta informació de manera còmode. Donat que l'aplicació no en disposa de cap, en aquest punt s'analitza la implementació d'una sobre el sistema.

4.2.2.1 Estat actual

Actualment, *Alice* no realitza cap mena d'ordenació dels objectes d'un model utilitzant criteris de distribució espacial. En el graf d'escena (descrit al punt 4.2.1.1), únicament es disposa d'una jerarquia que agrupa objectes que tenen una certa relació entre ells, però que només s'utilitza a efectes de selecció. En el cas de la visualització, s'utilitza una altra jerarquia, molt més simple, que considera tots els objectes (nodes *B-Rep*) al mateix nivell.

D'aquesta manera, el procés de visualització és incapaç de fer cap mena de distinció entre el tractament d'aquells objectes que té situats molt a prop i aquells que estan més lluny.

4.2.2.2 Alternatives considerades

En la literatura, apareix una gran varietat d'estructures jeràrquiques de diferents característiques. A continuació, es presenta una selecció dels tipus que s'han considerat i s'expliquen les principals propietats de cadascuna [7]:

a. *BSP (Binary Search Partition)*: Consisteix en un arbre que subdivideix recursivament l'espai, de manera que a cada nivell es fa una bisecció d'aquest a partir d'un pla de tall arbitrari. Cada pla de tall, secciona la regió corresponent en dos subespais. La divisió finalitza quan un subespai satisfà una determinada condició (ex. conté un determinat nombre d'objectes).

   Donat que l'element de divisió és un pla arbitrari, la selecció d'aquests esdevé un procés relativament complex. Tot i això, es disposa de més llibertat a l'hora d'ajustar-se a l'escena. Habitualment, s'utilitza algun tipus d'heurística que permet escollir aquells plans que fan que l'arbre resultant sigui òptim, i a la vegada, es minimitzin les interseccions amb objectes de l'escena.

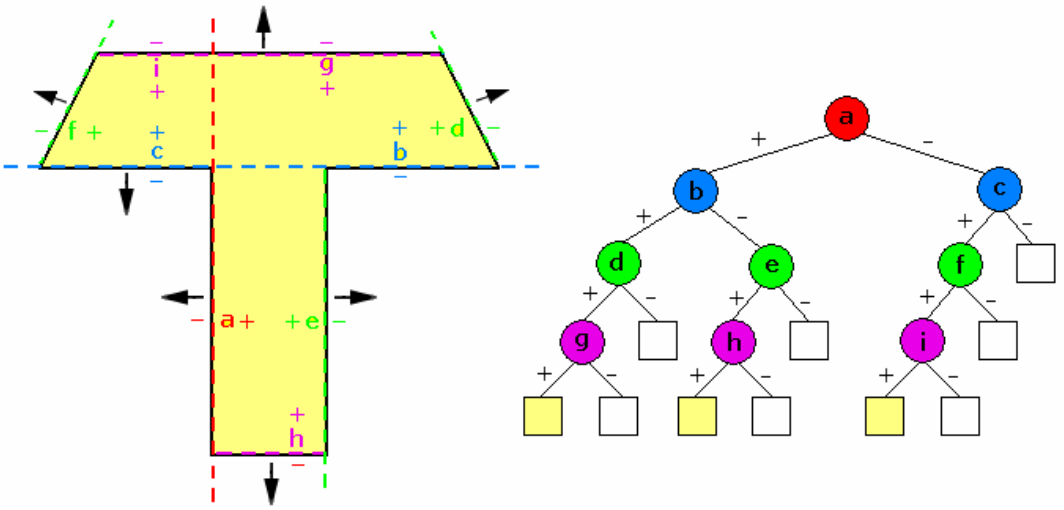


Fig. 4.2.2.2.1: *Exemple d'arbre BSP d'un políedre.*

b. <u>*K-d tree:*</u> Es tracta d'un arbre binari, molt similar als *BSPs*, on a cada nivell de recursivitat es subdivideix la regió en dos subespais mitjançant un pla. La principal diferència però, es troba en que els plans utilitzats són paral·lels als eixos de coordenades. Aquests van alternant la direcció del tall (X, Y o Z) segons el nivell de l'arbre en que es trobin. D'aquesta manera, hi ha menys graus de llibertat a l'hora de fer la divisió de l'escena, cosa que facilita el procés de divisió, però treu part de la capacitat d'ajustar-se a aquesta.

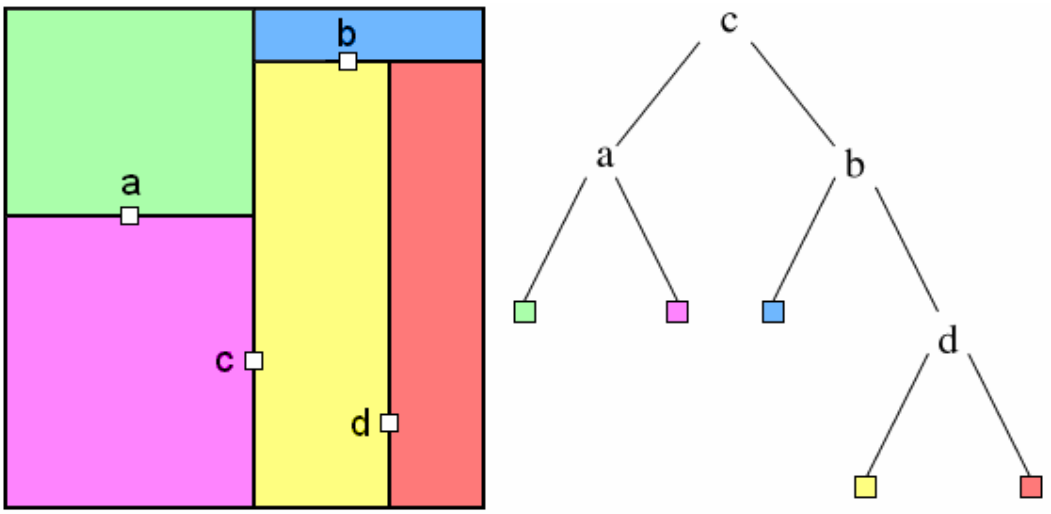


Fig. 4.2.2.2.2: *Representació d'un K-d tree en dues dimensions.*

c. *Octree:* És una estructura jeràrquica que recursivament divideix la regió de l'espai corresponent en vuit cubs d'idèntiques dimensions. Així doncs, a cada nivell es defineixen 3 plans de tall, alineats segons els eixos de coordenades (XY, XZ i YZ) i que passen pel punt mig de la regió de l'espai que s'està dividint. D'aquesta manera, s'obté un procediment que no requereix cap tipus de direcció de la subdivisió, però que tampoc té en compte les característiques de l'escena. Això pot provocar que hi hagin un gran nombre d'objectes que són tallats per algun d'aquests plans.

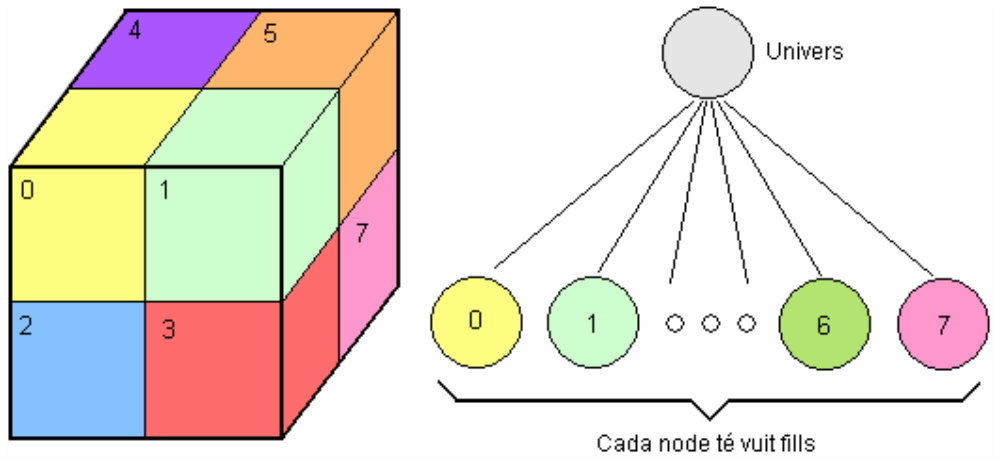


Fig. 4.2.2.2.3: *Representació d'un octree. Cada node es divideix en 8 regions.*

En les estructures anteriors, es pot observar clarament que la facilitat de construcció i l'adaptabilitat a l'escena són característiques diametralment oposades. Com que es tracta de dues propietats d'interès, cal escollir una opció que assumeixi un cert compromís entre aquestes. Per aquest motiu, es considera que el *K-d tree* és l'opció més adequada. La resta de l'anàlisi es basa en que aquest, és el tipus d'estructura jeràrquica que es vol implantar.

4.2.2.3 Principals discussions

En aquest apartat es mostren els aspectes més rellevants de la implementació d'un *K-d tree* sobre l'aplicació. Hi ha una sèrie de punts conflictius que cal resoldre. Són els següents:

1. *Ubicació de la nova estructura:* El *K-d tree* es situarà de forma paral·lela al graf d'escena que hi ha actualment. Això significa que els objectes continguts per cadascuna de les regions fulla del *K-d tree* correspondran a nodes *B-Rep* de l'*scene graph*. D'aquesta manera, es manté la dualitat entre les dues estructures i es permet accedir a la més adequada en cada cas. La figura 4.2.2.3.1 mostra esquemàticament aquesta nova organització:

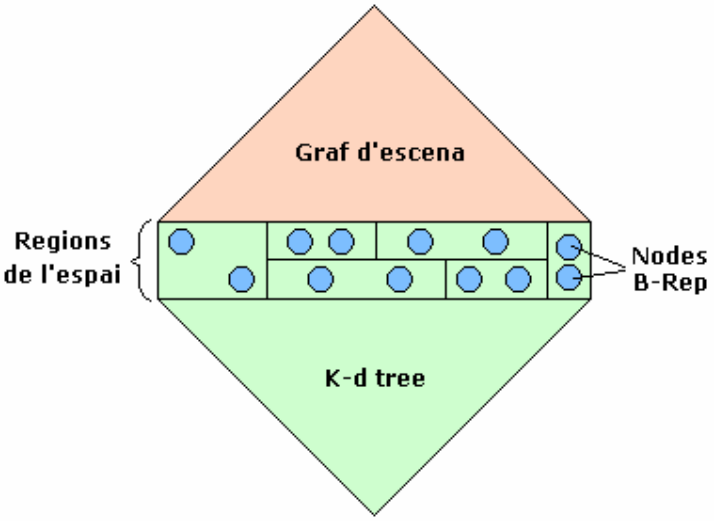


Fig. 4.2.2.3.1: *Esquema de la ubicació del K-d tree sobre l'estructura d'Alice.*

2. *Criteri de fulla*: La divisió s'aplica de forma recursiva mentre els nodes tenen un nombre d'objectes superior a un cert llindar. Per determinar el valor d'aquest llindar (criteri de fulla), cal tenir en compte quines de les altres optimitzacions plantejades faran ús d'aquesta estructura.

   Una de les propostes que preveu utilitzar-la és la dels *VBOs* (punt 4.1.1). En aquesta, es fa servir una estructura jeràrquica per a agrupar objectes de l'escena que es troben pròxims entre si i després generar els *VBOs* segons els grups que s'hagin format. S'estima que el tamany òptim d'un *VBO* és de 50.000 vèrtexs (tot i que és variable segons el tipus de placa). Per tant, d'entrada cal subdividir el *K-d tree* mentre es trobin nodes que tinguin més de 50.000 vèrtexs.

   La resta de millores que la fan servir són les *occlusion queries* (punt 4.3.1), la *jerarquia multiresolució* (punt 4.4.1), els *impostors* (punt 4.4.2) i les tècniques *out-of-core* (punt 4.5.1). En tots aquests casos, l'estructura jeràrquica s'usa per agrupar la geometria propera i per a permetre fer un

tractament individualitzat en cadascun dels grups obtinguts. La dimensió d'aquests grups ha de satisfer un equilibri entre la complexitat per a gestionar-los i l'adaptabilitat a les característiques del model. Cal doncs que que el cost de mantenir l'estructura quedi compensat amb els guanys obtinguts. L'única limitació existent es troba en el cas de les tècniques *out-of-core*, on es requereix que el tamany d'un node tingui cabuda en una pàgina de disc. D'aquesta manera, la mida determinada pel cas dels *VBOs* es considera vàlida i és la que s'adoptarà com a criteri de fulla de l'estructura.

3. *Gestió de les interseccions*: En algunes ocasions, el pla de tall utilitzat per subdividir un node pot intersecar amb algun objecte de l'escena. Això dificulta la classificació d'aquest objecte, ja que no queda ben definit en quin costat del pla es troba. Aquestes situacions es poden resoldre mitjançant un tipus de gestió especial. Existeixen dues possibilitats alternatives:

    a. *Duplicar objectes*: Consisteix en incloure els objectes intersecats en els dos nodes fills del que es divideix. Cal controlar molt bé la forma com es fa la visualització i així evitar enviar a pintar un mateix objecte dues vegades. Això, a part del innecessari consum de recursos, pot provocar greus problemes de *z-fighting*. A més a més, el consum de memòria, al haver-hi informació duplicada, augmenta notablement.

    b. *Tallar els objectes*: Es tracta de seccionar els objectes intersecats pel pla de divisió. D'aquesta manera, es substitueix l'objecte existent, per dos nous objectes formats per les primitives que queden a cada costat del pla de tall. Les primitives que intersequen són duplicades i associades als dos subespais generats. Aquesta opció requereix un major temps de càlcul per a la generació de l'estructura i incrementa el nombre d'objectes de l'escena. El nombre de primitives també es veu incrementat, però només una petita fracció del que augmenta amb l'altre opció. A més, una vegada construït el *K-d tree*, al haver-se tallat els objectes, no cal tornar a preocupar-se per aquest problema en cap dels processos que l'utilitzen.

Després d'aquest anàlisi, resulta evident que la opció més apropiada és la de tallar els objectes. Addicionalment, cal tenir present que la generació de l'arbre es correspon amb un preprocés i que per tant, no suposa un problema greu que aquest es vegi incrementat de forma raonable.

4.2.2.4 Pronòstic de rendiment

En aquesta ocasió, no s'hauria d'apreciar cap millora directa en el rendiment d'*Alice*. Tot i això, els canvis plantejats permeten desenvolupar moltes de les altres propostes que es descriuen. Per tant, cal redireccionar l'anàlisi cap als guanys obtinguts amb aquestes altres millores per a poder fer un pronòstic correcte.

4.2.2.5 Conflictes i interferències

Tot i que es tracta d'una requeriment per al desenvolupament de moltes altres millores, la incorporació d'una estructura jeràrquica no requereix ni entra en conflicte amb cap de les propostes estudiades.

4.3 Visibilitat

Un dels camps que més freqüentment s'ataca per tal d'accelerar aplicacions gràfiques és el de la visibilitat. Consisteix en determinar eficientment quines parts del model no son visibles i així estalviar-se haver d'enviar-les al *pipeline* gràfic. En el cas d'*Alice,* ja es troben implementades algunes d'aquestes tècniques: *back-face culling*, s'eliminen els polígons que no miren cap a l'observador, *frustum culling*, es descarta la geometria que es troba fora de *frustum* de visió, i *occlusion culling*, es defineixen manualment uns oclusors i s'elimina tot allò que queda tapat per aquests. En qualsevol cas, existeixen altres tècniques d'*occlusion culling*, la utilització de les quals pot incrementar encara més la velocitat del procés de visualització. A continuació, es dona una descripció de les principals i és fa un anàlisi sobre la seva implantació a l'aplicació.

4.3.1. Occlusion culling

Les tècniques d'*occlusion culling* consisteixen en detectar quins objectes queden completament tapats, per altres elements de l'escena, des d'un determinat punt de vista. Si s'aconsegueix fer aquesta detecció de forma eficient, es pot evitar haver de pintar els objectes oclusos, cosa que amb total seguretat, accelera notablement el *rendering*.

4.3.1.1 Estat actual

Els càlculs *d'occlusion culling* que actualment es fan a *Alice* es limiten a una sèrie d'oclusors definits manualment. L'usuari inspecciona l'escena i indica les coordenades d'uns polígons que creu que es corresponen millor als oclusors reals del model carregat. Així doncs, tots els nodes de *l'scene graph*, la caixa englobant dels quals queda completament tapada per aquests polígons, no són enviats al *pipeline* gràfic (veure figura 4.3.1.1.1).

Es tracta d'una tècnica molt poc adient ja que requereix de la participació de l'usuari per al seu correcte desenvolupament. Suposa una tasca molt feixuga i repetitiva, amb la qual cosa, la seva realització no acostuma a compensar en relació amb els beneficis que aconsegueix. A més, la qualitat i la precisió de les dades subministrades no estan garantides, i per tant, tampoc un correcte funcionament.

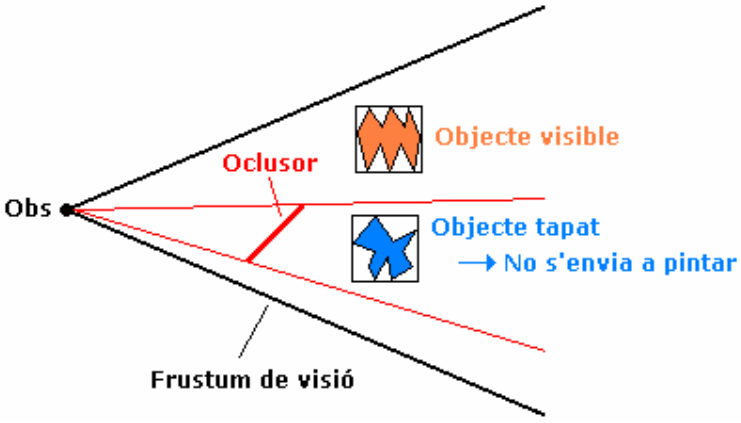


Fig. 4.3.1.1.1: *Occlusion culling mitjançant oclusors manuals.*

Per altra banda, en un intent de testejar les millores de rendiment aportades, s'ha provat de definir una sèrie d'oclusors utilitzant la interfície gràfica existent. Aquesta requereix la introducció de les coordenades numèriques dels punts dels polígons oclusors. Aquest mètode resulta molt poc intuïtiu i requereix molt d'esforç per part de l'usuari per col·locar els punts amb un mínim d'exactitud. Addicionalment, durant les proves realitzades, s'han detectat nombrosos errors fatals que forcen el final de l'execució de l'aplicació.

4.3.1.2 Alternatives considerades

Existeix un gran nombre de tècniques que, tot i ser de més complexitat que la que s'està usant, permeten realitzar *occlusion culling* de forma més sofisticada. A continuació, s'expliquen les que s'han tingut en compte per aquest projecte [8]:

a. *Aspect graph*: Es tracta de segmentar l'espai de punts de vista, de manera que s'agrupin en una mateixa regió tots els punts que tenen la mateixa visibilitat (veuen als mateixos objectes). La informació de com es configuren les diferents regions de visibilitat, per una escena determinada, s'emmagatzemen mitjançant l'*aspect graph*. Es tracta d'un mètode excessivament complex (té un cost computacional de $O(n^9)$), cosa que en fa inviable la seva utilització a *Alice*.

b. *Càlcul del PVS (Potential Visibility Set)*: Consisteix en dividir l'escena en regions (per exemple, utilitzant una estructura jeràrquica de divisió de l'espai) i en precalcular quins objectes són visibles des de cadascuna d'aquestes regions. S'assumeix que la visibilitat de tots els punts d'una regió és la mateixa. A més, els càlculs es fan de forma conservativa per evitar cometre errades en la imatge resultant.

   Aquesta tècnica té molt bon rendiment si el tamany de les regions que s'han creat és suficientment petit. En aquest cas, es permet estalviar el *rendering* de molta geometria i el cost addicional que cal pagar és despreciable. Tot i això, requereix un preprocés molt lent, molt complex i que utilitza una gran quantitat de memòria per emmagatzemar els resultats. A més, com que es tracta d'un precàlcul, no es permet cap mena de modificació ni moviment sobre els objectes de l'escena, cosa que inhabilita algunes de les operacions existents a *Alice.*

c. *Hardware occlusion queries*: Verifiquen la visibilitat d'un objecte a partir d'enviar una representació simplificada d'aquest, com ara un volum englobant, al hardware gràfic. La geometria es rasteritza i es compara amb el contingut del *z-buffer*. Sense modificar el contingut de cap buffer, el hardware retorna el nombre de pixels que superen el *z-test*. Així doncs, si no hi ha cap píxel afectat (o un nombre molt petit), l'objecte no és visible i per tant, no cal que es renderitzi.

   Es caracteritzen per la seva eficiència, donat que exploten la paral·lelització en la seva implementació. Tradicionalment, ja existia una extensió d'*OpenGL* que permetia aquest tipus de consultes (*HP_occlusion_test*). Aquesta però, no permetia executar múltiples *queries* de forma simultània i per tant, calia esperar la finalització d'una abans de llançar l'execució de la següent. En aquest marc, el cost de les consultes podia resultar més elevat que els propis beneficis que ofereixen. Això deixa de ser així amb l'aparició de l'extensió *NV_occlusion_query*. Aquesta aprofita la potència de les *GPUs* actuals per eliminar aquesta limitació i permetre utilitzar-les en càlculs de visibilitat.

   A més, les *hardware occlusion* queries tampoc imposen cap restricció sobre la forma dels objectes amb què tracten. Donat que els tests es fan

a nivell de fragment (*z-buffer*), poden tenir una forma completament arbitrària. Finalment, cal destacar que en aquesta ocasió, tots els càlculs es fan en temps d'execució. Això significa que es podrà editar l'escena sense que els càlculs de visibilitat obtinguin resultats erronis.

Després d'aquest primer anàlisi de possibilitats, s'ha considerat que les *hardware occlusion queries* són l'opció més indicada. Proporcionen molt bons resultats, en termes d'eficiència, i la complexitat d'implantar-les no és excessiva.

4.3.1.3 Principals discussions

La utilització de les *hardware occlusion queries* en el procés de visualització d'*Alice* té una sèrie d'aspectes que cal resoldre. Tot seguit, es comenten quins són i quina és la solució que es preveu adoptar en cada cas:

1. *Geometria proxy*: Les *occlusion queries* basen el seu funcionament en una representació simplificada de la geometria que es vol comprovar si és visible o no. Per una banda, interessa que aquesta s'ajusti el màxim possible a l'objecte real, per tal d'obtenir un cert grau de precisió en el test. Per l'altra, es recomana que no sigui complexa i així s'eviti condicionar negativament el cost de la consulta. Existeixen les següents opcions:

    a. *Bounding Sphere:* Consisteix en l'esfera mínima centrada en l'objecte, tal que conté tota la geometria d'aquest en el seu interior. És la representació més simple (només requereix un centre i un radi), però no s'ajusta massa bé als objectes. A més, algunes operacions que s'usen habitualment, com ara la unió, acostumen a generar esferes molt ineficients.

    b. *AABB (Axis Aligned Bounding Box)*: Es tracta d'utilitzar un prisma alineat als eixos i perfectament ajustat a l'objecte associat. És l'opció més idònia per a satisfer aquest compromís: aproximen relativament bé la geometria i a la vegada són molt simples (consten de només 6 polígons) i fàcils de calcular. Addicionalment, ja es troben implementades en els nodes de l'*scene graph* d'*Alice*, cosa que n'afavoreix encara més la utilització. Finalment, cal destacar que s'ajusten perfectament a la forma de les regions que generen la

majoria d'estructures jeràrquiques de divisió de l'espai i per tant, es poden conjuntar perfectament.

c. *OOBB (Object Oriented Bounding Box)*: Són molt similars a les *AABB's*, però amb la diferència que l'orientació de la capsa no es troba restringida als eixos de coordenades. D'aquesta manera, es permet una millor adaptació a les característiques de l'objecte contingut amb la mateixa complexitat geomètrica. Tot i això, aquests nous graus de llibertat compliquen considerablement els càlculs necessaris i les operacions entre *OOBBs* tampoc són gaire eficients.

d. *K-Dop*: Fa una aproximació de l'objecte contingut a partir de *k* direccions predeterminades. Es defineix un pla per cadascuna d'aquestes direccions, el conjunt dels quals delimita la regió en què consisteix el *k-dop*. De fet, quan k=4, el resultat coincideix amb una *AABB*. Com és evident, si s'amplia el nombre de direccions, augmenta l'adaptabilitat a l'objecte. Per contra, la complexitat (nombre de cares) i el cost dels càlculs es veuen afectats de forma negativa. En el cas de les *occlusion queries*, no es considera que l'augment de precisió obtingut compensi en relació amb la pèrdua de simplicitat de la representació.

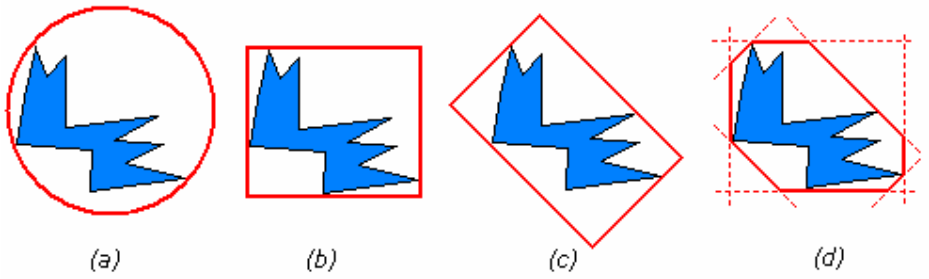


Fig. 4.3.1.3.1: *Diferents alternatives de geometria proxy.*
(a) Bounding Sphere (b) AABB (c) OOBB (d) 8-Dop al pla

Les quatre possibilitats exposades s'ordenen de més a menys simples, però també de menys a més precises. Tal i com s'ha dit, es decideix adoptar la opció de les *AABB,* situada en un punt intermedi. Així, s'aconsegueix un nivell de precisió acceptable amb uns càlculs relativament simples i a més, es permet fer una partició disjunta de l'espai.

2. *Organització de les queries*: Cal determinar quines consultes es volen efectuar i en quin ordre. Per tal d'aprofitar la paral·lelització que ofereix la nova extensió de *Nvidia*, es recomana llançar primer totes les consultes i després recollir-ne els resultats. D'aquesta manera, es descarta ja d'entrada el clàssic mètode *stop & wait*. Tot i això, es consideren dues formes d'organitzar les *queries* a realitzar:

    a. *Queries de tots els nodes*: Es tracta de llançar una *hardware occlusion query* per cadascun dels nodes del graf d'escena que contenen geometria (nodes *B-Rep*) i que es troben dins del frustum de visió. Una vegada llançades totes les consultes, es recullen progressivament els resultats i en cas que els nodes siguin visibles, s'envien a pintar. És molt fàcil d'implementar, però no té en compte la localitat espacial (nodes propers probablement tindran tots la mateixa visibilitat).

    b. *Coherent Hierarchical Culling*: És l'organització del mètode proposat a [9] i consisteix en fer ús d'una estructura jeràrquica de divisió de l'espai per a distribuir les consultes. Es tracta de fer un recorregut *top-down* per l'estructura existent, de manera que a cada node visitat es llanci una *hardware occlusion query* de la caixa englobant. En cas que el resultat sigui no visible, aleshores es podrà descartar el node i tots els seus descendents. En cas contrari, caldrà continuar el recorregut per cadascun dels fills. Finalment, quan s'arribi a una fulla i la consulta corresponent indiqui que l'objecte és visible, s'enviarà a pintar.

    Cal destacar que tot i fer-se un recorregut per l'arbre, les consultes es llancen en paral·lel. Així doncs, mentre s'espera el resultat d'una *query* en una branca, es realitzen tasques en una altra. Per altra banda, també s'optimitza el nombre de consultes a realitzar. S'emmagatzema el front de nodes de l'arbre que indica on aquests deixen de ser visibles (veure la figura 4.3.1.3.2). D'aquesta manera, a cada frame, només cal actualitzar el front i per tant, no s'ha d'iniciar cada vegada el recorregut des de l'arrel.

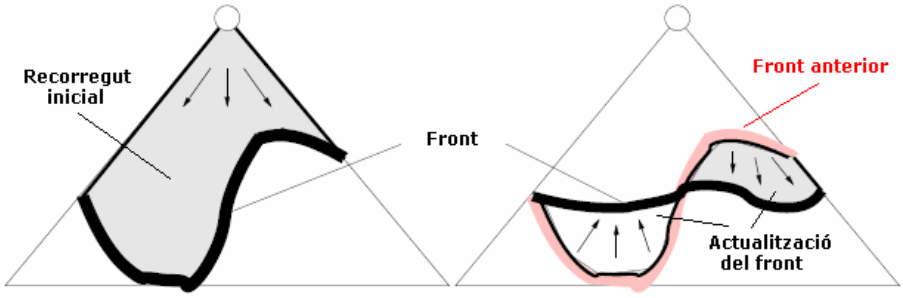


Fig. 4.3.1.3.2: *Front de nodes de l'arbre on aquests passen de visibles a no visibles.*

Tal i com s'ha pogut veure, l'algoritme *Coherent Hierarchical Culling* fa un millor ús de les *hardware occlusion queries* (tot i ser més complex i requerir una estructura jeràrquica). Per aquest motiu, es creu que aquest és el mètode que s'hauria d'implementar.

3. *Nombre de fragments llindar*: El resultat de les *occlusion queries* consisteix en el nombre de fragments rasteritzats que han superat el test de profunditat. Cal fixar un valor llindar que indiqui quants fragments són necessaris per a considerar que un objecte és visible. Si es volgués ser conservatiu, amb un únic fragment visible ja s'hauria de renderitzar l'objecte. Tot i això, donat que la geometria utilitzada en el test és una aproximació i per tant, els resultats no són totalment exactes, es tolerarà un cert error a l'hora de discriminar la visibilitat d'un objecte. Es fixarà, de forma empírica, un valor llindar (petit, però superior a 1) que determinarà el grau de tolerància permès. Així s'aconseguiran evitar situacions en les que l'esforç necessari per a pintar correctament uns pocs pixels és excessiu.

   També s'ha considerat la possibilitat de modificar aquest llindar en funció de la velocitat de navegació. Quan l'usuari s'està desplaçant, es requereix un *framerate* molt elevat per tal que la interacció sigui suau, però en canvi, la qualitat de les imatges generades pot ser lleugerament relaxada (no es posa massa èmfasi en el detall). En canvi, quan l'usuari està quiet, inspeccionant amb cura una zona, es produeix la situació inversa. D'aquesta manera, modificar la qualitat en funció de la navegació resulta una proposta interessant i a més, és fàcil de portar a terme. Tot i això, aquestes variacions poden generar *artifacts*. És el cas d'un o d'uns pocs pixels que constantment estan canviant la seva il·luminació (entre correcte i incorrecta). Això pot provocar que l'usuari, al detectar aquestes variacions,

hi centri la seva atenció, mentre que si s'hagués mostrat sempre la representació incorrecte, probablement no se'n hauria donat compte. Per aquest motiu, es desestima aquesta alternativa i es fixa el llindar com un valor constant.

4.3.1.4 Pronòstic de rendiment

La millora de rendiment que pot aportar aquesta optimització es veu clarament condicionada per la relació entre el cost d'execució de les *hardware occlusion queries* en relació amb el nombre d'objectes que deixen de renderitzar-se. Com és evident, això és un fet que depèn de les característiques de l'escena i en particular, del grau d'oclusió que hi hagi. En qualsevol cas, en comparació amb el tipus d'*occlusion culling* que s'està realitzant actualment, els guanys estan garantits i per tant, aquest és un sòlid candidat per implementar.

4.3.1.5 Conflictes i interferències

El desenvolupament proposat entra en conflicte o requereix d'algunes de les altres optimitzacions proposades en aquest treball. A continuació, es destaquen les principals interferències:

1. *Estructura de divisió de l'espai (punt 4.2.2)*: Donat que es volen organitzar les *occlusion queries* segons l'esquema *Coherent Hierarchical Culling* proposat a [9], esdevé imprescindible una estructura jeràrquica. L'article no restringeix el tipus d'estructura a utilitzar i per tant, un *K-d tree*, el tipus considerat en el punt 4.2.2, és un esquema perfectament vàlid.

2. *Impostors i jerarquia multiresolució (punts 4.4.1 i 4.4.2):* Es tracta de dues millores que simplifiquen el procés de visualització al preu de renderitzar els objectes amb una qualitat menor (els impostors ho fan de manera més extrema que la jerarquia multiresolució). Per tal que aquesta pèrdua de qualitat no s'aprecií en el resultat final, només s'utilitzen aquestes tècniques en aquells objectes que es projecten en uns pocs pixels (un nombre inferior a un cert llindar). Així doncs, cal sincronitzar aquests llindars amb el de les *occlusion queries* per tal que cada tècnica s'apliqui segons un rang de pixels adequat, però sense anul·lar-ne cap.

3. *Out-of-core (punt 4.5):* Les tècniques proposades en aquest punt consideren no tenir en memòria tots els nodes de l'escena. Això entra directament en conflicte amb els càlculs de visibilitat, ja que part de la informació necessària pot trobar-se a disc. D'aquesta manera, caldrà sincronitzar els dos desenvolupaments per tal que aquest tipus de situacions no es produeixin.

4.4 Multiresolució

Les estratègies de multiresolució consisteixen en tenir diverses representacions dels objectes de l'escena, cadascuna amb un nivell de detall diferent. D'aquesta manera, l'aplicació fa servir la representació que considera més adient segons les necessitats que té a cada moment. Per exemple, un ús típic consisteix en seleccionar el grau de precisió dels objectes en funció de la distància a la que es troben de l'observador. Així doncs, tot i que actualment *Alice* ja considera l'ús de la multiresolució, aquest punt tracta sobre la implantació de noves tècniques relacionades amb aquest àmbit i la millora de les existents.

4.4.1 Jerarquia multiresolució

L'estructura actual d'*Alice* considera la possibilitat de tenir, per un mateix objecte, múltiples representacions en diferents nivells de detall (*LOD's*). Tot i això, aquestes representacions es troben definides de forma individual sobre cadascun dels objectes i per tant, no es té en compte cap tipus de distribució espacial a l'hora de seleccionar la representació més adient. D'aquesta manera, l'objectiu d'aquest apartat consisteix en implantar la construcció d'una jerarquia multiresolució sobre *Alice*, i així disposar d'informació que permeti fer una visualització més acurada.

4.4.1.1 Estat actual

L'scene graph que implementa actualment *Alice* és capaç de gestionar múltiples representacions d'un mateix objecte de l'escena. Això ho fa gràcies als nodes *LOD*, on cadascun dels seus fills equival a una d'aquestes representacions. D'aquesta manera, en el moment de renderitzar aquest node, es selecciona la representació més convenient i s'envia a pintar únicament la geometria que es troba sota la branca escollida (veure la figura 4.4.1.1.1).

La construcció dels nodes *LOD*, igual que la resta del graf d'escena, es fa en el moment de carregar el model. Això significa que els diferents nivells de detall de cada objecte ja han d'estar emmagatzemats en el fitxer *.P3D*. En un gran nombre d'ocasions però, només hi ha guardada una única representació i per tant, no es pot utilitzar la multiresolució. Així doncs, resultaria interessant modificar el procés

de lectura de models, per tal de crear representacions simplificades dels objectes en els casos que no n'hi hagi.

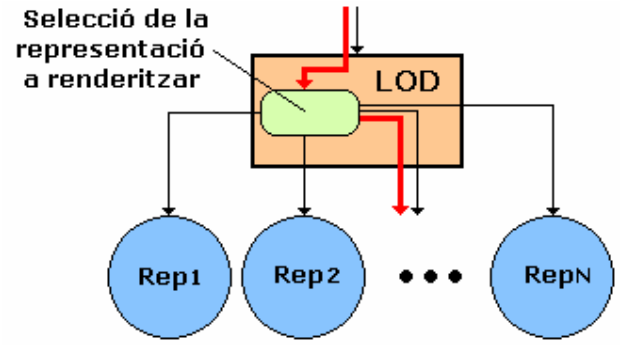


Fig. 4.4.1.1.1: *Renderitzat de un objecte amb múltiples representacions.*

El procés de selecció de la representació a renderitzar és molt simple. Utilitza la distància entre l'observador i l'objecte (prenent-ne el centre de la caixa englobant) com a únic criteri. Així doncs, aquells que es trobin més lluny utilitzaran representacions amb poca resolució, mentre que els que estiguin més a prop en faran servir de més detallades. Tot i això, existeixen situacions, com és el cas d'objectes allargats, en que no s'escull el *LOD* més convenient. Per resoldre-ho, s'haurien de considerar altres factors que poden ajudar a fer aquesta discriminació de forma més adequada.

4.4.1.2 Alternatives considerades

El sistema multiresolució d'*Alice* té algunes deficiències que cal resoldre. Per fer-ho, es proposa incorporar una jerarquia multiresolució, de manera que les diferents representacions s'associïn als nivells de profunditat de l'estructura jeràrquica de divisió de l'espai utilitzada. Existeixen dues possibles estratègies:

a. *Objectes independents*: Es tracta de mantenir una estructura similar a l'actual, en la qual es tenen múltiples representacions per un mateix objecte generades a partir de les *Quadric Error Metrics (QEM)* [10]. En aquest cas però, cada representació també s'associa a un únic node de l'estructura jeràrquica disponible. En els nodes més profunds, hi ha representacions més detallades mentre que en els més pròxims a l'arrel n'hi ha de més simplificades.

Presenta com a principal avantatge la simplicitat. Permet aprofitar gran part de l'estructura existent i es pot integrar fàcilment a l'aplicació. Tot i això, la complexitat topològica és manté constant al llarg de tots els nivells de precisió: hi ha sempre el mateix nombre d'objectes. Això suposa un problema a l'hora de simplificar molts objectes pròxims entre ells. Al tractar-los individualment, l'error produït és major que si es fes de forma global i per tant, la qualitat de les imatges obtingudes pot ser menor.

b. *Fusió d'objectes*: També consisteix en construir els diferents nivells de detall segons una estructura jeràrquica, però en aquesta ocasió, a més a més, es preveu fusionar objectes pròxims entre si [11]. Així, també es podrà reduir la topologia de l'escena a cada nivell de detall. D'aquesta manera, cal modificar tot el procés de generació dels *LODs* per tal que simplifiquin els objectes de forma global (els continguts en una mateixa regió de l'escena) i no de forma individual, tal i com es fa actualment (veure la figura 4.4.1.2.1).

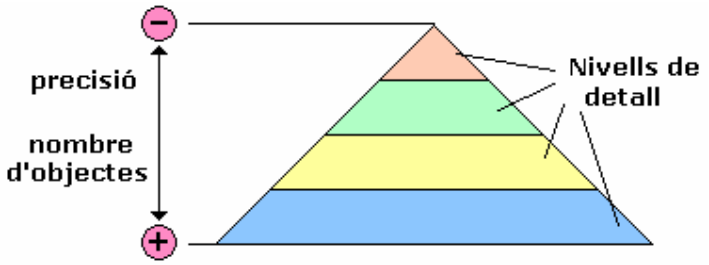


Fig. 4.4.1.2.1: *Esquema d'una jerarquia multiresolució amb fusió d'objectes.*

L'impacte de portar a terme aquesta proposta implica canvis de certa importància. Cal construir les representacions a partir d'utilitzar nous tipus de tècniques de simplificació, com ara les volumètriques i això fa que calgui redissenyar completament el procés existent. Tot i la dificultat afegida, en aquest cas es resol el problema de la complexitat topològica. Es permeten obtenir representacions simplificades de més qualitat, però a la vegada, poden aparèixer *artifacts* més notables al passar d'un nivell de precisió a un altre.

El principal avantatge que aporta la fusió d'objectes respecte al tractament independent és la simplificació de la topologia de l'escena. En molts casos però, això suposa un problema degut a *artifiacts* de *cracking* provocats als canviar de representació. Per aquest motiu, es desestima la fusió d'objectes i s'opta per l'altra estratègia, molt més simple i que permet aprofitar part del mecanisme existent a *Alice*.

4.4.1.3 Principals discussions

Els principals focus discussió que s'han considerat en la implantació d'una jerarquia multiresolució sobre *Alice* són els següents:

1. *Nombre de nivells de detall*: Cal fixar la granularitat, segons el grau de precisió, amb la que es construeixen representacions simplificades dins la jerarquia. En un cas ideal, es generarien representacions diferents per tots els nivells de profunditat de l'arbre. Això donaria una adaptabilitat màxima (hi hauria un ampli ventall de representacions a escollir), però el consum de memòria per emmagatzemar-les seria excessiu. D'aquesta manera, es decideixen fer agrupacions de nivells de profunditat consecutius i crear una única representació per cada grup. Així s'aconsegueix un estalvi important en la memòria consumida i a la vegada es conserva una gran adaptabilitat.

   Per altra banda, cal destacar que esdevé complicat establir una relació entre la complexitat geomètrica de l'escena i la seva distribució espacial. Per aquest motiu, es fixa el nombre de representacions que es generaran per cada objecte, sempre i quan es superi un cert llindar de complexitat (nombre de triangles). Aquest és un valor que s'hauria de determinar de forma empírica.

2. *Criteri de selecció*: Es pretén utilitzar un mecanisme de selecció més sofisticat, però igualment eficient, i així utilitzar la representació més adequada en cada cas. Segons [12], hi ha fins a 6 criteris que s'haurien de considerar per fer aquesta elecció: dimensions de l'objecte en pantalla (nombre de pixels projectats), error introduït en les simplificacions (en relació amb el nombre de polígons), importància (definida per l'usuari), focus (distància al centre de la pantalla), moviment (relació entre la velocitat de l'objecte i la mida) i histeresis (intenta mantenir el mateix nivell de detall que en el frame anterior).

Així doncs, a partir de tots aquests indicadors es pretén construir un heurístic que indiqui quin és el benefici que té cada objecte en la visualització. D'aquesta manera, es pot executar un algoritme de *rendering* basat en un pressupost [12], és a dir, sense que es superi un cert nombre màxim de polígons per frame: s'utilitza la combinació de representacions que maximitza el benefici total i no excedeix el pressupost. S'ha de dir que fer el càlcul exacte pot resultar molt costós, fins al punt d'eliminar completament els beneficis de la multiresolució. Per aquest motiu, s'adopta una estratègia *greedy*, molt ràpida i que genera combinacions molt properes a l'òptima.

3. *Esquema de construcció:* La creació de la jerarquia multiresolució que s'ha previst, utilitza un esquema basat en una estructura jeràrquica de divisió de l'espai. El procés consta de dos fases:

    1. *Fase top-down*: A partir d'un recorregut per l'arbre, des de l'arrel fins a les fulles, es distribueix tota la geometria de l'escena (en la seva representació més detallada) entre els diferents nodes fulla. De fet, això és justament el mateix que es fa en la pròpia construcció de l'estructura jeràrquica. Per tant, no cal realitzar accions addicionals per completar aquesta fase.

    2. *Fase bottom-up*: En un altre recorregut, aquest cop ascendent, s'assigna a cadascun dels nodes interiors una representació dels objectes continguts en la regió representada, fins arribar a l'arrel. El nivell de detall de les representacions utilitzades disminueix progressivament cada cop que es puja un nombre prefixat de nivells.

    Cal destacar que aquest esquema no simplifica l'escena des d'un punt de vista global, sinó que ho fa a partir d'atacar petits trossos de malla. Això fa viable la construcció de la jerarquia multiresolució en el cas de models gegantins, el tractament dels quals es comenta en el punt 4.5.1.

4.4.1.4 Pronòstic de rendiment

El temps de rendering disminuirà principalment en aquells models que prèviament no tenien multiresolució, i s'hi ha incorporat gràcies a aquesta optimització. Tot i això, la resta de models també acceleraran la seva visualització

degut a un ús més eficient de les múltiples representacions. Es preveu maximitzar la qualitat de les imatges obtingudes, en base a un pressupost que garanteix que l'aplicació funcionarà en temps real.

4.4.1.5 Conflictes i interferències

El desenvolupament d'aquesta millora té alguns punts estretament relacionats amb algunes de les altres optimitzacions proposades. Concretament, es tracta de les següents:

1. *Estructura de divisió de l'espai (punt 4.2.2)*: Es proposa organitzar les diferents representacions de cada objecte segons una estructura jeràrquica d'aquest tipus. Per tant, és imprescindible haver-ne implementat una abans de prosseguir amb aquest desenvolupament.

2. *Occlusion Culling* i *impostors (punts 4.3.1 i 4.4.2)*: Si s'implementa alguna d'aquestes millores caldrà ajustar les toleràncies dels criteris de selecció per tal que s'utilitzi la representació més adient en cada cas.

3. *Out-of-core (punt 4.5)*: Cal considerar com gestionar correctament la multiresolució en el cas de models molt grans. S'ha de determinar quines representacions s'emmagatzemen en memòria principal i quines es guarden a disc. També es necessita adaptar convenientment el mecanisme que gestiona la transferència de dades entre memòries.

4.4.2 Utilització d'impostors per a geometria llunyana

Un tipus de simplificació que permet representar fidelment objectes molt llunyans són els impostors. Tot i que n'hi ha una extensa varietat, aquests es basen en substituir geometria que es troba a una certa distància per una sèrie de polígons texturats. D'aquesta manera, es fa una combinació entre geometria, utilitzada per als elements pròxims, i impostors, empleats per als elements llunyans. Cal destacar que les imatges utilitzades pels impostors acostumen a generar-se durant un preprocés. Per tant, cal seleccionar molt bé quins són els que es volen fer servir, ja que no es preveu crear-ne de nous en temps d'execució. Així doncs, en aquest apartat es dona una descripció dels principals trets respecte com s'hauria d'implementar aquesta tècnica a *Alice*.

4.4.2.1 Estat actual

En aquests moments, l'aplicació només és capaç de visualitzar geometria. De fet, a excepció de les representacions simplificades dels nodes *LOD*, tot el que s'utilitza consisteix en la geometria real del model que es troba dins del *frustum*. Això significa que els objectes més allunyats, que es projectin sobre uns pocs pixels del pla de projecció, també requereixen enviar una gran quantitat de geometria al *pipeline* gràfic per a visualitzar-los.

Per altra banda, tampoc es disposa de cap estructura per a gestionar impostors. El graf d'escena d'*Alice* només té nodes per a emmagatzemar políedres (*B-Rep)* o bé nodes que representen imatges *2D* situades sobre el pla de projecció (*Picture2D*).

4.4.2.2 Alternatives considerades

La literatura proposa una tipologia d'impostors molt àmplia i variada. A continuació, es mostren els tipus que s'han considerat, juntament amb els avantatges i inconvenients de cadascun:

a. *Polígons texturats*: Es tracta d'incloure polígons (triangles, quadrilàters, ...) texturats amb imatges generades en un preprocés i que capturen una representació de la part de l'escena que substitueixen (veure la figura 4.4.2.2.1) [13]. Són molt fàcils de construir, però donat que es tracta d'una imatge estàtica, no suporten efectes de *parallax*. Això significa que només són vàlids per un conjunt reduït de punts de vista i que per tant, introdueixen errors amb facilitat.

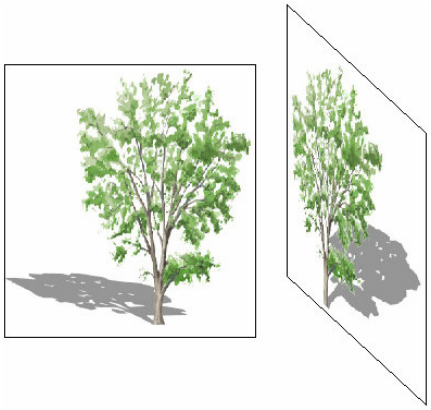


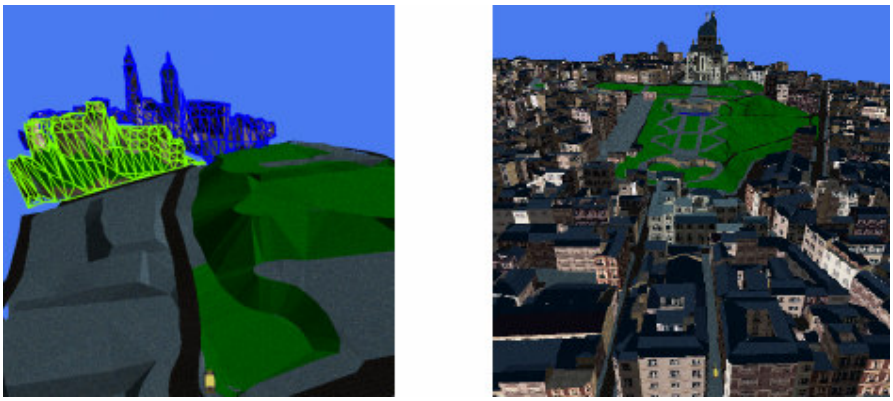
Fig. 4.4.2.2.1: *Impostor basat en un polígon texturat. Segons el punt de vista, apareixen artefactes degut a la falta d'efectes de parallax.*

b. *Billboard clouds:* [14] Es simplifiquen els objectes a partir de descomposar-los en un conjunt de plans que contenen textures i mapes de transparència (veure la figura 4.4.2.2.2). D'aquesta manera, s'aconsegueix que puguin ser vàlids des de qualsevol punt de vista i acoten el nivell màxim d'error que produeixen. Tot i això, el temps necessari per a generar-los és molt alt i si no s'usa un nombre de reduït de plans, tendeixen a produir artefactes.

(a) *(b)* *(c)*

Fig. 4.4.2.2.2: *Impostor de tipus billboard cloud.*
(a) Polígons on es col·loquen els billboards (b) Textures utilitzades (c) Imatge resultant

c. *Malles de triangles texturades:* [15] Consisteix en representar la geometria que es troba més allunyada de l'observador mitjançant trossos de malla de triangles, que utilitzant uns pocs polígons, capturen els principals trets dels elements que representen. Es tracta doncs, d'una tècnica situada a mig camí entre els impostors i les representacions simplificades. Al tractar-se de malles, produeixen efectes de parallax i per tant, prolonguen el camp de visió des del qual són vàlids. Tot i això, la seva construcció i funcionament acostuma a estar pensat per a escenes de certes característiques (ex. ciutats) i no per a models de caràcter general.

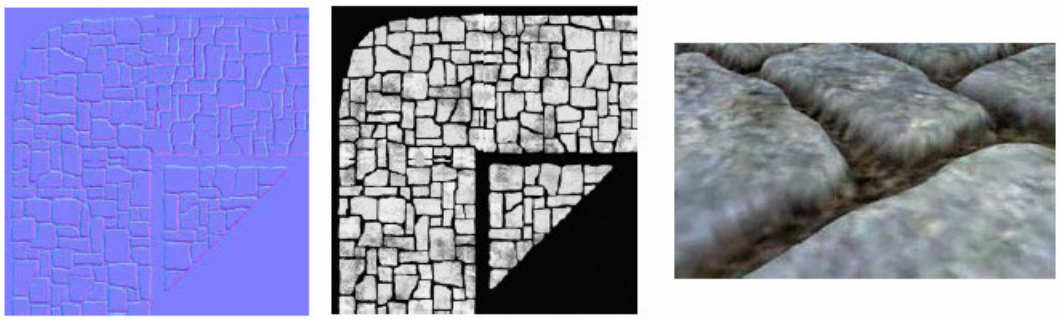


(a) *(b)*

Fig. 4.4.2.2.3: *Impostor basat en una malla de triangles texturada.*
(a) Malles de triangles que representa geometria llunyana (b) Imatge resultant

d. *Relief impostors:* [17] Són polígons texturats mitjançant *relief mapping*, una tècnica que utilitzant un mapa de normals, un mapa de profunditats i una textura, permeten simular superfícies amb relleu (veure la figura 4.4.2.2.4). Estan basats en un paradigma de traçat de raigs i implementen la seva visualització mitjançant *shaders* (molt eficient pels càlculs que es fan). Obtenen imatges de molta qualitat, que suporten efectes de *parallax* i mostren les ombres produïdes per les auto-oclusions. Tot i això, la quantitat de memòria de *GPU* necessària es veu notablement incrementada (cal emmagatzemar mapes addicionals) i el procés de visualització, tot i ser molt eficient, és més costós. Addicionalment, caldria fer correccions addicionals per tal d'evitar artifacts quan s'utilitzés aquesta tècnica en sistemes que disposen de visió estereoscòpica.

   Una especialització interessant dels *relief impostors*, són els *ORIs* [18] (*ommnidirectional relief impostors*). Consisteixen en la combinació de múltiples *relief impostors*, situats estratègicament per tal de representar correctament a un objecte des de qualsevol punts de vista.



(a) *(b)* *(c)*

Fig. 4.4.2.2.4: *Relief impostor.*
(a) Mapa de normals (b) Mapa de profunditats (c) Imatge resultant

Tot i que s'han analitzat propostes que obtenen resultats de millor qualitat, s'acaba optant pels polígons texturats degut a l'extrema senzillesa que tenen. Es creu que si es combinen amb multiresolució, només es faran servir per a elements molt distants a l'observador i per tant, no es requereix un nivell de qualitat excessiu. Així doncs, en els següents punts es pren el supòsit de que aquest, és el tipus d'impostor que s'utilitzarà.

4.4.2.3 Principals discussions

La utilització d'una tècnica basada en impostors sobre *Alice* té alguns aspectes que cal acabar de definir. Tot seguit, es presenten els més importants:

1. *Mètode de posicionament*: Els polígons texturats es mantenen vàlids en un rang relativament reduït. Tenint present aquest fet, cal especificar un mètode que defineixi quins impostors es volen utilitzar i com s'han de situar. Aquests es generen durant un precàlcul, previ a l'execució, i a partir de renderitzar parts de l'escena des de diferents posicions i orientacions de l'observador.

   La solució proposada consisteix en dividir l'escena en cel·les amb forma paral·lelepípede. Per cadascuna d'aquestes cel·les, es creen una sèrie d'impostors (6 textures rectangulars) que permeten cobrir-ne les parets, tal i com mostra la figura 4.4.2.3.1. Així, quan una cel·la es trobi a una certa distància de l'observador i es projecti en un petit nombre de pixels en pantalla, es pot substituir la geometria continguda en el seu interior pels impostors generats.

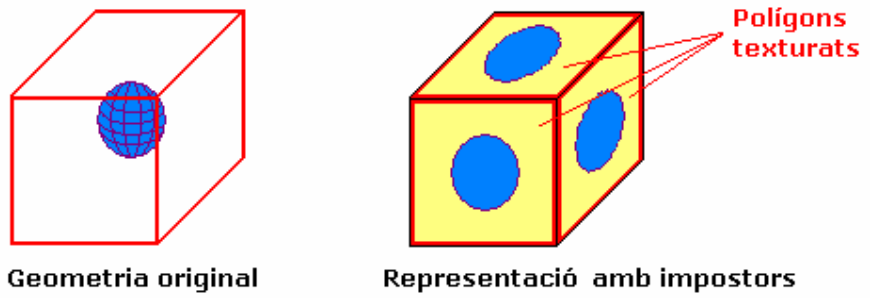


Fig. 4.4.2.3.1: *Impostors que cobreixen les parets de cadascuna de les cel·les de l'escena.*

Per tal d'efectuar la divisió de l'escena en cel·les, es pretén utilitzar la partició de l'espai en regions que proporciona una estructura jeràrquica de divisió de l'espai (punt 4.2.2). D'aquesta manera, s'aconsegueix que la geometria que representa cada grup d'impostors sigui aproximadament constant.

A més a més, per a poder crear impostors amb diferents graus de precisió, es planteja utilitzar les regions dels diferents nivells de profunditat de l'arbre de l'estructura jeràrquica. Així doncs, si un node pare es projecta amb un nombre de pixels reduït, aquest ja es renderitza directament amb impostors i per tant, no cal que es navegui pels seus fills per a visualitzar-lo.

Finalment, cal destacar que si es posicionen els impostors d'aquesta manera, el màxim error es produirà quan l'observador estigui mirant directament alguna de les arestes de la cel·la (veure la figura 4.4.2.3.2). Donat que aquest tipus d'impostors no gestionen efectes de *parallax*, s'han de tenir molt presents aquests casos per poder evitar que l'usuari detecti *artifacts*.

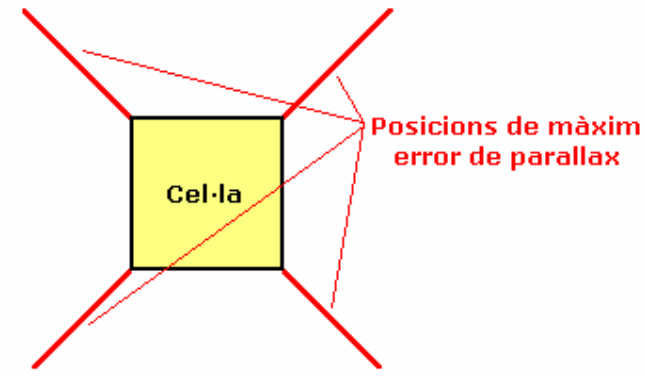


Fig. 4.4.2.3.2: *Situació de màxim error quan es visualitza una cel·la mitjançant impostors.*

2. *Criteri de selecció:* La visualització d'un node de l'estructura jeràrquica mitjançant impostors es determina a partir de la petjada deixa aquest en el *viewport*. Quan el tamany d'aquesta petjada sigui es inferior a un cert llindar, el node es renderitza utilitzant els impostors disponibles.

L'assignació d'un valor a aquest llindar ha de buscar un equilibri entre una maximització de l'ús d'impostors i la minimització de la pèrdua de qualitat que es produïda. Si els impostors només es fan servir ocasionalment, els guanys obtinguts seran mínims. Per contra, si el llindar permet utilitzar-los amb nodes molt grans, l'error produït serà major i els *artifacts* de *cracking* poden veure's incrementats. Així doncs, s'haurà de

fixar aquest valor de forma empírica, fent que aquest el màxim de gran possible, però sense que s'apreciïn *artifacts* destacables.

3. *Resolució de les textures*: Donat que s'utilitzaran per a renderitzar polígons que tenen uns pocs pixels a la pantalla, interessa que siguin de baixa resolució, a efectes de minimitzar la memòria consumida per a emmagatzemar-les. Tot i això, el seu valor està directament relacionat amb el llindar descrit en el punt anterior. A més, s'estima que hauran de tenir una resolució una mica superior a la que fixa aquest llindar. Així es podrà tenir una mica més de precisió que permetrà evitar errors al distorsionar-la per a texturar els polígons.

4.4.2.4 Pronòstic de rendiment

L'augment de rendiment que aquesta optimització preveu aportar està molt condicionat pels tipus d'escena que s'utilitzin i per les altres millores que s'implementin. Els resultats òptims s'obtenen en el cas d'escenes obertes i espaioses, sense gaires oclusions, però amb una gran quantitat d'objectes simples. En aquests casos, les tècniques de visibilitat no poden eliminar gaires elements mentre que la multiresolució té uns efectes limitats. Així, la utilització d'impostors influencia molt positivament el rendiment final de la visualització. En altres situacions però, el solapament amb altres millores és més important i com a conseqüència, el benefici obtingut serà menor.

4.4.2.5 Conflictes i interferències

L'ús d'impostors com a mètode per a representar geometria llunyana interfereix amb algunes de les altres millores presentades. A continuació, es comenten els principals punts de conflicte:

1. *Estructura de divisió de l'espai (punt 4.2.2)*: El sistema de posicionament dels impostors està estrictament lligat a una estructura d'aquest tipus. D'aquesta manera, resulta imprescindible tenir-ne una de implementada abans d'iniciar aquest desenvolupament.

2. *Jerarquia multiresolució (punt 4.4.1)*: Es tracta d'una millora que actua sobre el mateix camp que els impostors. Tot i que entren directament en

conflicte, es considera que poden coexistir sense problemes: utilitzar multiresolució per als objectes relativament propers i impostors per als que es troben molt distants de l'observador. En qualsevol cas, s'hauran d'ajustar molt acuradament els criteris que permeten decidir quina és la tècnica que s'utilitza a cada moment.

3. *Gestió de memòria (punt 4.5)*: La incorporació d'impostors requereix espai addicional per al seu emmagatzemament. Així doncs, caldrà tenir-los en compte els impostors a l'hora de fer canvis en la gestió de la transferència d'informació.

4.5 Gestió de memòria

Tot sistema de visualització mínimament potent utilitza múltiples tipus de memòria per a emmagatzemar els models que es carreguen. La idea consisteix en tenir en la memòria més ràpida (la de la targeta gràfica), tota la informació que es necessita per a *renderitzar* l'escena i part de la que es preveu utilitzar pròximament. Tot i això, la capacitat de la *GPU* és limitada i en moltes ocasions cal enregistrar dades en memòries més lentes, però de major capacitat. És el cas de la memòria principal (*RAM*) o fins i tot, quan es treballa amb models gegantins, del disc. Així doncs, per tal que aquest fet no afecti negativament al rendiment de l'aplicació, cal cuidar molt la forma en com es mou la informació entre memòries. Per tant, aquest punt s'ocupa de proposar millores que permetin gestionar de manera eficient aquestes transferències de dades.

4.5.1 Visualització de models gegantins

El tamany màxim del models amb què pot treballar l'aplicació ve determinat per la quantitat de memòria *RAM* disponible. Així doncs, no resulta possible carregar models gegantins que superin aquesta xifra, com pot ser el cas de representacions extremadament detallades o espais molt grans. Per tal de resoldre aquesta limitació, existeixen unes tècniques, anomenades *out-of-core*, que permeten treballar amb part del model emmagatzemat a disc i part en memòria. D'aquesta manera, es tracta de gestionar correctament la transferència d'informació per tal que sempre hi hagi a memòria la part del model que es vol visualitzar. En aquest punt, es proposa la implantació d'alguna d'aquestes tècniques per a permetre que *Alice* sigui capaç de suportar models de dimensió arbitrària.

4.5.1.1 Estat actual

Actualment l'aplicació no permet visualitzar models que excedeixin la capacitat màxima de la memòria *RAM*. Concretament, el que es fa és carregar a memòria principal tots els nodes del graf d'escena que hi ha enregistrats en el fitxer *.P3D*. D'aquesta manera, quan no hi ha suficient memòria per a guardar tots els nodes, l'aplicació és incapaç d'obrir el model. Això suposa una limitació destacable pel que fa a les funcionalitat que ofereix *Alice*. Existeix un gran nombre d'aplicacions, com és el cas del sector naval (d'interès per al projecte *BAIP 2020*), que habitualment

utilitzen models d'aquestes característiques (el disseny d'un vaixell sencer de grans dimensions, com per exemple, un portaavions) i si no se'ls permet llegir-los, significa que s'estan deixant de banda.

4.5.1.2 Alternatives considerades

Per tal de poder gestionar aquest tipus de models, es considera la utilització d'una memòria alternativa: el disc. Es tracta d'un espai de gran capacitat (permet emmagatzemar diversos *Gigabytes*), tot i que té una velocitat d'accés menor a la de la memòria *RAM*. En qualsevol cas, es desitja mantenir a memòria principal la part del model que es visualitza i la que es preveu utilitzar en els pròxims *frames*. La resta d'informació s'hauria de guardar a disc i en instants previs al moment en que passi a ser requerida, enviar-la a memòria principal.

S'ha d'anar amb molta cura amb la manera amb com s'organitza el flux de dades entre memòries. Cal transferir la informació, amb suficient antelació, de forma que estigui disponible abans que l'algoritme de *rendering* la necessiti. Així, no s'haurà d'esperar la elevada latencia de la transmissió i com a conseqüència, es podrà evitar que el rendiment de l'aplicació es vegi perjudicat.

D'aquesta manera, una gestió eficient de la transferència d'informació entre aquestes dues memòries consisteix en pronosticar correctament quins objectes passaran a visualitzar-se. Per fer-ho, únicament es proposa la utilització d'una versió ampliada del frustum de visió. Això consisteix en prendre el frustum actual, extendre'l una mica i determinar quins elements de l'escena es troben dins seu (figura 4.5.1.2.1). Com que la navegació acostuma a ser continua en l'espai, els objectes que es troben a la part afegida del frustum són els que, amb alta probabilitat, passaran a renderitzar-se (són consecutius als que ho fan actualment).

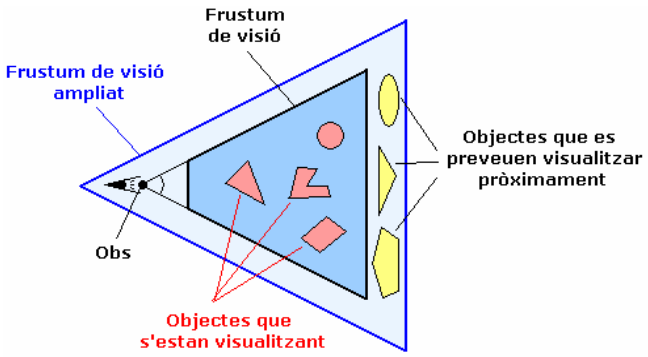


Fig. 4.5.1.2.1: *Ús d'un frustum ampliat per a pronosticar els pròxims objectes que es visualitzaran.*

4.5.1.3 Principals discussions

A continuació, es presenten els principals punts de discussió que es plantegen a partir del desenvolupament d'aquesta proposta:

1. *Extensió del frustum*: Es vol calcular una versió ampliada del *frustum* de visió per a poder pronosticar quins objectes es visualitzaran en els propers *frames*. S'ha de tenir en compte que com més gran sigui la magnitud del nou *frustum*, més gran serà el nombre d'objectes continguts i per tant, hi haurà més probabilitats de portar a memòria els objectes necessaris. Tot i això, l'espai de memòria que s'utilitzarà també serà major. Així doncs, cal assumir un compromís, el valor del qual dependrà de la quantitat de memòria *RAM* disponible.

   Per altra banda, també es suggereix ajustar la forma del *frustum* segons la tipologia del mode de navegació actiu. La idea consisteix en ampliar-lo en major mesura en les direccions que habitualment es prenen amb el mode de navegació seleccionat, mentre que en les direccions que no s'utilitzen mai o gairebé mai, l'extensió hauria de ser menor. D'aquesta manera, s'aconsegueix fer una previsió més acurada, i per tant, més eficient dels objectes que es visualitzaran

2. *Transmissió d'informació*: Es tracta d'un aspecte molt delicat, ja que una mala gestió pot originar una baixada de rendiment important. Així doncs, el moment idoni per a realitzar la transferència de dades és quan la *CPU* es troba lliure, sense càrrega de treball. Això succeeix quan la *GPU* està pintant l'escena, ja que la *CPU* està a l'espera de que acabi. Per tant, es pretén aprofitar aquest lapse de temps per a actualitzar convenientment la memòria principal i així poder fer que aquesta tasca sigui transparent a l'aplicació.

   A més, es proposa separar l'algoritme de *rendering* pròpiament dit i el mecanisme de gestió de memòria mitjançant *multithreading*. Es planteja crear un *thread* per cada activitat, de manera que es puguin explotar els múltiples nuclis que acostumen a tenir els ordinadors moderns. El punt 4.6.1 descriu de manera més detallada com es porta a terme aquesta paral·lelització.

Finalment, resta pendent definir el tamany dels enviaments que es realitzin. Interessa que aquests siguin de valors pròxims al tamany d'una pàgina de disc o múltiples d'aquest tamany. D'aquesta manera, s'aconsegueix que les transferències siguin òptimes, ja que la informació s'envia en paquets d'aquestes dimensions. Tot i que resulta difícil ajustar aquest valor, el que si que es pot fer és mirar d'anticipar el pas a memòria d'alguns elements i així omplir completament les pàgines de disc.

3. *Visualització excessiva*: Pot donar-se la situació en la que es visualitzi de manera simultània un nombre d'objectes enorme, de manera que no càpiguen en memòria principal. Això acostuma a produir-se quan es treballa amb un model gegantí i es mira el model des de l'exterior, de forma que tots els elements d'aquest es trobin dins el frustum de visió. Si no es fa cap tipus de tractament que eviti aquest tipus de casos, l'aplicació presentarà problemes evidents de funcionament. Caldran fer, a cada *frame*, diverses transferències entre la memòria principal i el disc, cosa que reduirà espectacularment el *frame rate*.

   Per a poder resoldre aquest problema, es planteja utilitzar el sistema multiresolució disponible, ja sigui una jerarquia de representacions simplificades o un conjunt de cel·les amb impostors (veure propostes del punt 4.4). D'aquesta manera, esdevé un requisit indispensable tenir algun sistema d'aquestes característiques.

4.5.1.4 Pronòstic de rendiment

El desenvolupament d'aquesta optimització no suposarà, en cap cas, una acceleració en el rendiment global d'*Alice*. Això es justifica gràcies al fet que es preveu passar a explotar una memòria més lenta que les que s'utilitzen actualment. Tot i això, es considera necessària ja que s'espera que elimini la limitació del tamany dels models carregats, cosa que permetrà fer que l'aplicació sigui accessible a un ventall d'usuaris molt més ampli.

4.5.1.5 Conflictes i interferències

La visualització de models gegantins és una millora que modifica constantment la disponibilitat de la informació en les diferents memòries. Per aquest motiu, té un

impacte molt fort en la majoria de les propostes que es fan. Tot seguit, es fa un comentari sobre com caldria adaptar les altres optimitzacions afectades:

1. *Vertex Buffer Objects (punt 2.1.1)*: Cal procurar tenir a memòria principal tots els *VBOs* que s'hagin de visualitzar. Això es pot garantir fàcilment gràcies a l'estratègia del *frustum* ampliat, la qual permet detectar ràpidament quines parts del model es necessiten.

2. *Estructura jeràrquica de divisió de l'espai (punt 2.2.2)*: Es tracta d'un tipus d'informació que moltes optimitzacions preveuen utilitzar amb freqüència i de forma global. A més, la quantitat de memòria que requereix tampoc és excessiva. D'aquesta manera, es creu que s'ha de mantenir tota l'estructura sempre en memòria principal. Tot i això, donat que els objectes referenciats no tenen perquè trobar-s'hi, es considera interessant incorporar a l'arbre apuntadors a la zona de disc on s'emmagatzemen. Pel que fa a la seva construcció, com que no es poden tractar totes les dades de cop, aquesta s'haurà de realitzar mitjançant algun dels mètodes *out-of-core* existents [19, 20, 21].

3. *Jerarquia multiresolució (punt 2.4.1)*: La combinació d'aquesta millora amb la tècnica out-of-core proposada es pot considerar des de dos punts de vista: mantenir en memòria totes les representacions d'un objecte carregat, o bé, només tenir-hi la versió que s'està utilitzant. La primera opció té una gestió més simple, mentre que la segona requereix menys memòria. Cal determinar quina és l'alternativa més adequada i adoptar-la. Per altra banda, també s'ha pogut veure que és una de les solucions per a evitar la situació no desitjada de la visualització excessiva.

4. *Ús d'impostors (punt 2.4.2)*: Es necessita un espai de memòria considerable per a emmagatzemar els impostors que s'utilitzin. Això significa que si es passa a guardar-los a disc, es podrà alliberar una quantitat important de memòria *RAM*. Tot i això, caldrà tenir en compte aquest fet a l'hora de gestionar la transferència d'informació, de manera que els impostors d'una determinada cel·la es carreguin en memòria principal quan els objectes continguts també ho facin. Finalment, cal dir que aquesta és una altra de les alternatives per a resoldre el problema de la visualització excessiva.

4.6 Multithreading

Una de les evolucions recents i més significatives en el hardware utilitzat pels computadors ha estat la utilització de múltiples processadors. D'aquesta manera, pràcticament tots els *PCs* que s'utilitzen actualment disposen de, com a mínim, dues unitats *CPU*. Tot i això, per poder-les explotar al màxim, cal que el programari que s'hi executi estigui paral·lelitzat, és a dir, que cal que es creïn diversos fils d'execució *(threads)* per a portar a terme diferents tasques de forma simultània. Així, cada *thread* es pot executar en un processador diferent. En el cas d'*Alice* però, ens trobem davant d'una aplicació *monothread*, i per tant, mentre un dels processadors s'ocupa de tots els còmputs, els altres no tenen res a fer. Això obre la porta al grup d'optimitzacions que es presenten en aquest apartat. A partir de paral·lelitzar part del codi de l'aplicació, es poden aconseguir rendiments molt més elevats.

4.6.1 Threads auxiliars per a processos de càlcul

L'*API d'OpenGL* no està dissenyada per a treballar amb sistemes amb múltiples fluxos d'execució. Per aquesta raó, resulta molt complicat fragmentar les tasques relacionades amb la visualització de manera que es processin de forma paral·lela. En particular, si que és possible portar-ho a terme, però cal anar amb molta cura, ja que s'han incorporar algunes proteccions addicionals per evitar problemes de concurrència. En qualsevol cas, no es tracta d'una pràctica gaire recomanable.

Per altra banda, cal destacar l'existència d'altres llibreries gràfiques, com és el cas de *OpenSG* [22], que faciliten notablement la utilització de *multithreading*. Tot i això, *Alice* és una aplicació basada en *OpenGL* i migrar-la a una altra plataforma suposaria un canvi de magnitud extrema i per tant, no és viable desenvolupar-lo.

D'aquesta manera, es pretén donar prioritat a altres canvis, relatius als processos de càlcul existents o als processos que es volen afegir amb altres optimitzacions. Es considera que paral·lelitzar aquest tipus de tasques és una opció més segura (menys susceptible a generar errors) i que també pot contribuir a accelerar el rendiment d'*Alice*.

4.6.1.1 Estat actual

El funcionament actual de l'aplicació és pràcticament *monothread*. Això significa que gairebé totes les tasques s'executen de forma seqüencial i que per tant, no hi ha molt poca paral·lelització. Així doncs, els processos de càlcul existents es desenvolupen en el mateix flux d'execució que la visualització.

De fet, l'únic ús que *Alice* fa del *multi-core* es troba en els precàlculs d'il·luminació. Com que no s'utilitza la il·luminació *d'OpenGL* (per raons històriques), es calcula el color il·luminat de cada vèrtex en temps de càrrega del model (cal tenir en compte que amb aquest mètode no es considera la component especular). Així doncs, per tal d'agilitzar la lectura del model, es mouen tots aquests còmputs en un thread auxiliar. D'aquesta manera, la visualització del model pot iniciar-se molt abans mentre que la il·luminació es va calculant de forma progressiva.

Addicionalment, també cal dir que els múltiples processadors també s'aprofiten quan s'executa de manera simultània amb altres aplicacions. En aquest cas, és el propi sistema operatiu qui gestiona la utilització dels recursos, assignant a cada procés un dels nuclis disponibles.

4.6.1.2 Alternatives considerades

Aquesta proposta consisteix en separar els processos que són purament de càlcul del que és el *rendering* de l'escena, de forma que es puguin desenvolupar simultàniament. Per dur a terme aquesta separació, es considera l'opció de crear diferents *threads*, de manera que cadascun s'encarregui d'una d'aquestes tasques. Així doncs, es presenta el següent esquema de fluxos d'execució:

1. *Flux principal*: És el responsable de la visualització de l'escena i com a conseqüència, de la interacció amb *OpenGL*. Bàsicament, el que ha de fer consisteix en executar les comandes adients, per tal d'enviar el model al *pipline d'OpenGL*. Per aconseguir-ho, ha d'utilitzar la geometria carregada i la informació calculada amb altres processos de l'aplicació.

2. *Fluxos auxiliars*: Cadascun dels processos de càlcul que s'identifiquin es correspondrà amb un nou flux d'execució. Així, es permetrà paral·lelitzar aquestes tasques amb el propi procés de *rendering*. Tot i això, caldrà

sincronitzar aquests *threads* amb el flux principal. Això és necessari per poder assegurar que les dades que comparteixen es mantinguin consistents.

D'aquesta manera, es tracta d'identificar quins processos de l'aplicació són possibles candidats a descompondre's en fluxos d'execució auxiliars. Cal analitzar tan els procediments ja existents com els que es preveuen incloure amb les altres millores d'aquest treball. A partir d'aquí, s'ha de valorar per quins d'ells suposa una bona millora deslligar-los del flux principal i per quins no. S'han detectat els següents casos:

1. *Selecció del nivell de detall*: Una de les tasques necessàries per poder utilitzar un sistema multiresolució, ja sigui l'existent o la proposta de millora del punt 4.4.1, és la selecció de la representació que es vol utilitzar per cada objecte. Això, si es vol fer de manera acurada i utilitzant molts criteris, es pot acabar convertint en un procés relativament costós. D'aquesta manera, esdevé un bon candidat per a separar-se de la visualització mitjançant un *thread* auxiliar. Es tractaria de calcular el *LOD* a usar per cada objecte en el proper frame mentre s'està fent el *rendering* de l'escena.

2. *Gestió dels VBOs*: En el cas que es decideixi implementar la optimització dels *VBOs* (punt 4.1.1) i es vulgui fer servir una política pròpia, diferent a la del *driver*, caldrà dissenyar un procés que mantingui a la memòria de la *GPU* els *VBOs* necessaris a cada frame. Aquest, haurà d'enviar a la targeta gràfica la geometria que convingui, segons la navegació que s'està duent a terme. En aquesta ocasió, això també es pot desvincular del procediment pintat de l'escena creant un nou flux i així es podria executar de manera simultània amb aquest.

3. *Gestió de models out-of-core*: La millora exposada en el punt 4.5.1 parla sobre el tractament de models gegantins mitjançant memòria externa (disc). Si finalment es decideix implantar, la principal dificultat es troba en gestionar eficientment la transferència d'informació, de manera que a memòria principal hi hagin sempre les dades que es visualitzen. Així doncs, es considera la possibilitat de deslligar aquesta gestió, mitjançant un altre *thread auxiliar*, com una alternativa perfectament vàlida.

4. *Construcció d'estructures de dades*: Alguns dels apartats d'aquest anàlisi d'alternatives suggereixen la creació o la modificació d'estructures de dades.

Es tracta de l'estructura jeràrquica de divisió de l'espai (punt 4.2.2), de la jerarquia multiresolució (punt 4.4.1) i del sistema d'impostors (punt 4.4.2). En tots els casos, es dona l'opció de construir-les durant la pròpia lectura del model o bé, en una aplicació independent que desenvolupa el preprocés necessari. Si s'escollís la primera opció, totes aquestes tasques de generació es podrien ubicar en un nou flux. Tot i això, es desestima aquesta possibilitat perquè la visualització no es pot iniciar fins que no s'ha completat la construcció d'estructures (el *rendering* en fa ús). Per tant, la paral·lelització obtinguda seria nul·la i com a conseqüència, no s'obtindria cap tipus de benefici en el rendiment de l'aplicació.

Per tant, es considera la separació del flux principal dels tres primers processos de la llista anterior. D'aquesta manera, el sistema operatiu podrà gestionar el desenvolupament dels procediments, de manera que s'executin de forma paral·lela i utilitzant els múltiples processadors disponibles.

4.6.1.3 Principals discussions

Hi ha una sèrie d'aspectes que cal acabar de precisar sobre el desenvolupament que es descriu en aquest punt. Són els següents:

1. *Sincronització dels fluxos*: La utilització de *multithreading* implica l'execució paral·lela de moltes de les tasques definides. Tot i això, aquestes tasques no són idèntiques i per tant, requereixen quantitats de temps diferents per a completar-se. A més, cal destacar que s'utilitzen dades compartides. D'aquesta manera, les operacions s'han d'executar en un ordre determinat per a obtenir el resultat correcte. Per aconseguir-ho, fa falta sincronitzar els diferents fluxos. Això consisteix en definir una sèrie de punts (punts de sincronització), on els diferents processos s'esperen. Així es permet obtenir l'ordre d'execució que es desitja.

   Donat que es tracta d'un problema típic, els propis sistemes operatius ja tenen implementats mecanismes de sincronització: semàfors, mutex, ... Addicionalment, si la portabilitat del sistema és un requeriment important, es poden utilitzar llibreries, com ara *Boost* [23] o *GNU Portable Threads* [24], que també ofereixen aquest tipus de funcionalitats.

Així doncs, per assolir sincronisme entre els diferents processos definits, cal considerar que els fluxos auxiliars bàsicament preparen la informació que s'utilitzarà en el *rendering* del següent *frame*. Així, el flux de visualització haurà d'esperar a que els *threads* auxiliars acabin abans d'iniciar el pintat de la propera imatge. Per altra banda, els procediments auxiliars no poden començar a fer càlculs fins que el *thread* de *rendering* no hagi llegit la informació que generen. Així doncs, en l'esquema de la figura 4.6.1.3.1, es pot veure un exemple d'execució de la sincronització que s'ha d'obtenir:

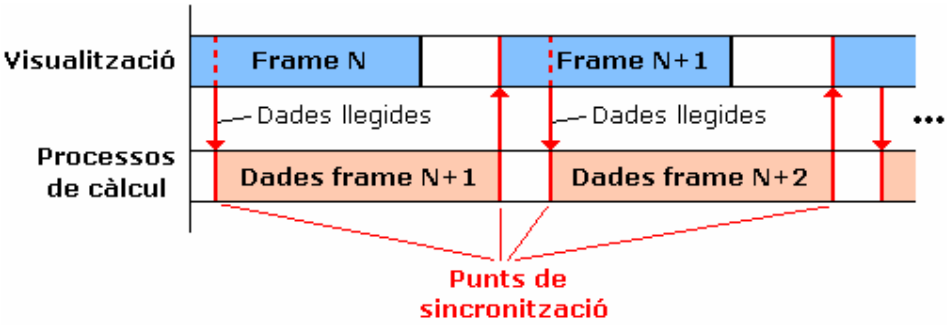


Fig. 4.6.1.3.1: *Esquema de sincronització seguit pels threads que hi haurà a l'aplicació.*

2. *Distribució de recursos:* Cal repartir el temps de *CPU* disponible entre els diferents *threads* de l'aplicació. Interessa organitzar-ho de manera que es minimitzi l'espera que fan els processos a l'hora de sincronitzar-se. En aquest punt, cal tenir en compte que no es fa cap supòsit sobre el grau de paral·lelisme del hardware disponible. Així doncs, no es pot conèixer a priori quants processadors hi haurà. Això significa que no sempre es podran assignar tots els fluxos a un processador diferent i que per tant, caldrà ajustar millor la divisió de recursos.

D'aquesta manera, per tal d'aconseguir una segmentació més acurada, es proposa utilitzar el mecanisme de prioritats existent. La majoria de llibreries de *threads* permeten assignar un valor de prioritat a cadascun dels fluxos existents. Aquest valor l'utilitza l'algoritme de *scheduling* del sistema operatiu per a distribuir el temps de càlcul entre els fils de l'aplicació. Així doncs, es pot anar modificant aquest valor per tal d'accelerar els processos que urgeixin més, a canvi de perjudicar els que no corren tanta pressa.

Tot i això, caldrà efectuar alguns experiments previs per a determinar les prioritats més adequades. S'utilitzarà una bateria de models de prova, suficientment àmplia, de manera que es pugui veure quines tasques habitualment costen més de desenvolupar. En base als costos obtinguts, es podrà estimar correctament el valor de la prioritat de les fases de cada procés existent.

4.6.1.4 Pronòstic de rendiment

La incorporació de múltiples fluxos d'execució permetrà paral·lelitzar moltes de les tasques que es desenvolupen durant el *rendering* d'un *frame*. D'aquesta manera, com més gran sigui el solapament aconseguit, major serà l'acceleració en el rendiment de l'aplicació. Per tant, es preveu un increment important, donat que es podrà eliminar gran part de l'elevat cost dels processos de càlcul. Tot i això, aquestes millores no es podran apreciar en màquines amb un únic processador. Com és evident, encara que es reparteixi la feina en múltiples fluxos, aquests s'hauran d'executar, obligatòriament, de manera seqüencial.

4.5.1.5 Conflictes i interferències

La major part de processos que es preveuen desvincular del flux principal apareixen gràcies al desenvolupament d'altres millores. Això significa que gran part dels beneficis que s'esperen obtenir requereixen que aquestes s'hagin implementat prèviament. En cas contrari, no té gaire sentit la implantació d'aquesta iniciativa, ja que els guanys obtinguts serien mínims. En particular, les relacions de dependència amb les altres optimitzacions d'aquest treball són: la utilització de *VBOs*, la construcció d'una jerarquia multiresolució i la gestió de models gegantins.

5 SELECCIÓ D'OPTIMITZACIONS A IMPLEMENTAR

La llista de millores exposada en el punt anterior és molt extensa i com és evident, no resulta possible (per restriccions de temps), ni té sentit (degut a interferències entre elles) implementar-les totes. Per aquest motiu, cal fer un valoració global de totes les propostes i seleccionar les que resultin més apropiades per a desenvolupar en aquest projecte. En els següents apartats, es fa una comparació de les diferents optimitzacions, es trien les més convenients i finalment, es distribueixen les tasques d'implementació al llarg del període que s'ha assignat a la planificació.

5.1 Valoració de les optimitzacions

Per poder valorar les diferents alternatives plantejades, es prenen tres criteris: el guany que s'espera per cadascuna d'elles, el temps previst d'implementació i les relacions de dependència existents. En les pròximes seccions, es mostren dades que qualifiquen totes les optimitzacions des d'aquestes tres perspectives.

5.1.1 Estimació del guany

A la descripció de les millores feta en el punt anterior, ja s'ha comentat el guany que s'espera obtenir amb cadascuna de les optimitzacions. La taula següent en mostra un resum:

Optimització	Guany esperat	Observacions
Vertex Buffer Objects	Alt	El coll d'ampolla de l'aplicació es troba en la transferència de dades cap a la placa gràfica.
Format dels fitxers P3D	Cap	Optimitza el temps de càrrega dels models, no el temps de visualització.
Estructura jeràrquica	Cap	Necessària per moltes altres millores.
Occlusion Queries	Mitjà	Milloren el sistema d'oclusors actual. Maximitzen el guany en escenes amb un grau d'oclusió molt gran.

Optimització	Guany esperat	Observacions
Jerarquia multiresolució	Mitjà	Es simplifica el tractament de parts de l'escena al preu de reduir la qualitat final. Pot produir artefactes.
Impostors	Mitjà	Efectes similars als de la jerarquia multiresolució, però més extrems.
Out-of-core	Cap	Permet tractar amb models gegantins.
Multithreading	Mitjà	Redueix molt el cost gràcies a l'execució en paral·lel de determinades tasques, però es focalitza en càlculs que no són crítics.

Taula 5.1.1.1: *Valoració del guany estimat per cada una de les optimitzacions.*

5.1.2 Estimació del temps d'implementació

Per tal de fer una planificació realista de les tasques d'implementació que es realitzaran, cal abans fer una previsió del temps necessari per desenvolupar les diferents possibilitats. A continuació, es desglossen les diferents optimitzacions en tasques i es dona una estimació del cost de cadascuna:

Nom de la tasca	Duració estimada
- Vertex Buffer Objects	**60 h**
Conversió de la geometria en malles de triangles	20 h
Operador de gestió de *VBOs*	10 h
Generació dels VBOs a partir de les malles de triangles	20 h
Modificació del procés de rendering per a utilitzar els nous *VBOs*	10 h
- Nou format dels fitxers P3D *	**60 h**
Modificació de les rutines de lectura i salvat	30 h
Aplicació per convertir al nou formats	30 h
- Estructura jeràrquica	**45 h**
Estructura de dades	10 h
Algoritme de selecció del pla de divisió	20 h
Gestió dels objectes fragmentats pel pla de divisió	15 h
- Occlusion Queries	**40 h**
Operador de gestió de les *hardware occlusion queries*	10 h
Algoritme de rendering basat en *Coherent Hierarchical Culling*	30 h

Nom de la tasca	Duració estimada
- Jerarquia multiresolució	**100 h**
Construcció de les representacions en diferents nivells de detall	80 h
Procés de selecció de la representació a renderitzar	20 h
- Impostors	**50 h**
Generació de les textures utilitzades en els polígons impostors	35 h
Algoritme que decideix quan s'han d'utilitzar els impostors	15 h
- Out-of-core	**105 h**
Construcció estructura jeràrquica amb tècniques out-of-core	30 h
Càlcul de la geometria que cal tenir a memòria principal	15 h
Gestió de la transferència de dades entre memòries	60 h
- Multithreading *	**45 h**
Separació en *threads* auxiliars dels principals processos de càlcul	25 h
Gestió de la concurrència i sincronització de fluxos	20 h
TOTAL	**505 h**

** Tasques subjectes a variabilitat degut a la interacció amb altres optimitzacions (es pot requerir temps addicional per a gestionar el conflicte).*

Taula 5.1.2.1: *Estimació de cost de les diferents alternatives proposades.*

Com es pot comprovar amb les dades de la taula, el temps disponible en aquest projecte (210 hores) és insuficient per a implementar-ho tot i per tant, seleccionar allò que és més representatiu, esdevé imprescindible.

5.1.3 Relacions de dependència / conflicte

Les interaccions entre millores també és una informació que ja s'ha proporcionat en el capítol previ. Tot i això, per poder tenir una visió global de les diferents relacions que s'estableixen, es pot observar el graf de dependències de la figura 5.1.3.1. Les connexions entre optimitzacions poden ser de tres tipus: de dependència (cal haver implementat totes les millores predecessores abans de desenvolupar una optimització), de dependència opcional (es requereix haver implementat almenys una millora predecessora) o d'interferència (s'han d'adaptar convenientment les propostes afectades per tal que no hi hagi conflicte).

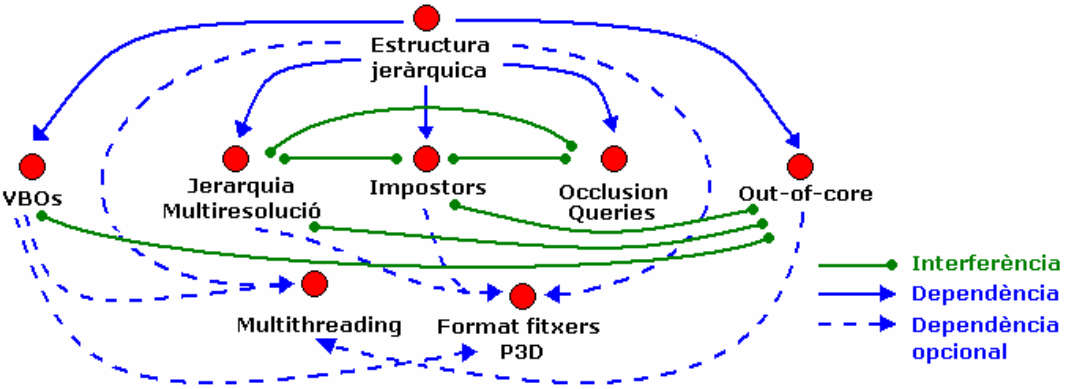


Figura 5.1.3.1: *Graf de dependències entre les diferents optimitzacions considerades.*

5.2 Presa de decisions

A partir del graf de dependències anterior, es pot veure clarament que per a poder desenvolupar satisfactòriament qualsevol de les propostes, resulta necessari haver implementat prèviament una *estructura jeràrquica de divisió de l'espai*. Per aquest motiu, es decideix que aquesta sigui la primera optimització que es porti a terme.

Tot seguit, s'opta per a continuar amb el renderitzat eficient de la geometria mitjançant *Vertex Buffer Objects (VBOs)*. En l'anàlisi on s'estimen els guanys, s'indica que aquesta millora ataca directament el coll d'ampolla de l'aplicació i que per tant, es preveu incrementar notablement el rendiment. A més a més, no hi ha dependències insatisfetes que la bloquegin i el temps d'implementació previst és molt raonable.

Amb l'elecció d'aquestes dues alternatives, es preveu consumir unes 105 hores (45h. de l'estructura jeràrquica + 60h. dels *VBOs*) de les 210 hores disponibles en total. D'aquesta manera, es disposa de 105h. addicionals per a treballar en alguna de les propostes restants.

D'entrada, es desestima el canvi a un *nou format dels fitxers P3D* i el *multithreading* degut a les dependències (tot i que opcionals) amb les altres millores. Aquestes obtenen el seus majors beneficis quan s'han implementat totes les optimitzacions predecessores i per aquest motiu, es deixen per a futures fases del projecte *BAIP 2020*.

Pel que fa a l'ús de tècniques *out-of-core*, malgrat que aquestes permetrien treballar amb models molt més grans, requereixen fer canvis importants a l'estructura de l'aplicació. Això es tradueix en un gran nombre d'hores de programació, que tot i ser factible abordar-les en aquest projecte, no dona marge per a l'error. Addicionalment, la gestió de models en memòria externa té fortes vinculacions amb les altres millores que encara no s'han escollit. Tot això, fa que s'opti per atacar en primera instància les alternatives relatives al temps de visualització i que es deixi com l'*out-of-core* com una possible ampliació.

Així doncs, de totes les propostes restants (*occlusion queries*, *impostors* i *jerarquia multiresolució*) s'espera obtenir una acceleració de característiques similars. Per això es decideix implementar en primera instància les *hardware occlusion queries*, ja que són les que suposen un esforç de desenvolupament menor. D'aquesta manera, encara es preveu que sobri un nombre suficient d'hores (105h. - 40h = 65h.) per poder atacar una quarta optimització. S'escull la que fa referència a *impostors*, donat que la *jerarquia multiresolució* no resulta viable amb el temps disponible. Les hores restants es deixen com a marge de seguretat per a possibles errors en les estimacions realitzades.

5.3 Planificació del desenvolupament tècnic

Una vegada seleccionades les propostes que es desenvoluparan, cal determinar com s'organitzaran al llarg del temps. Per temes de dependències, esdevé imprescindible que la incorporació d'una *estructura jeràrquica de divisió de l'espai* sigui el primer que es faci. A partir d'aquí, és indiferent l'ordre amb que es desenvolupin les altres tres propostes. De fet, la relació d'interferència entre les *occlusion queries* i els *impostors* només té a veure amb el llindar que defineix el criteri d'utilització de cadascuna. D'aquesta manera, es decideix començar amb les *occlusion queries*, seguir amb els *VBOs* i acabar amb els *impostors*. En el diagrama de *Gantt* de la figura 5.3.1 mostra la distribució temporal que s'ha escollit de forma gràfica:

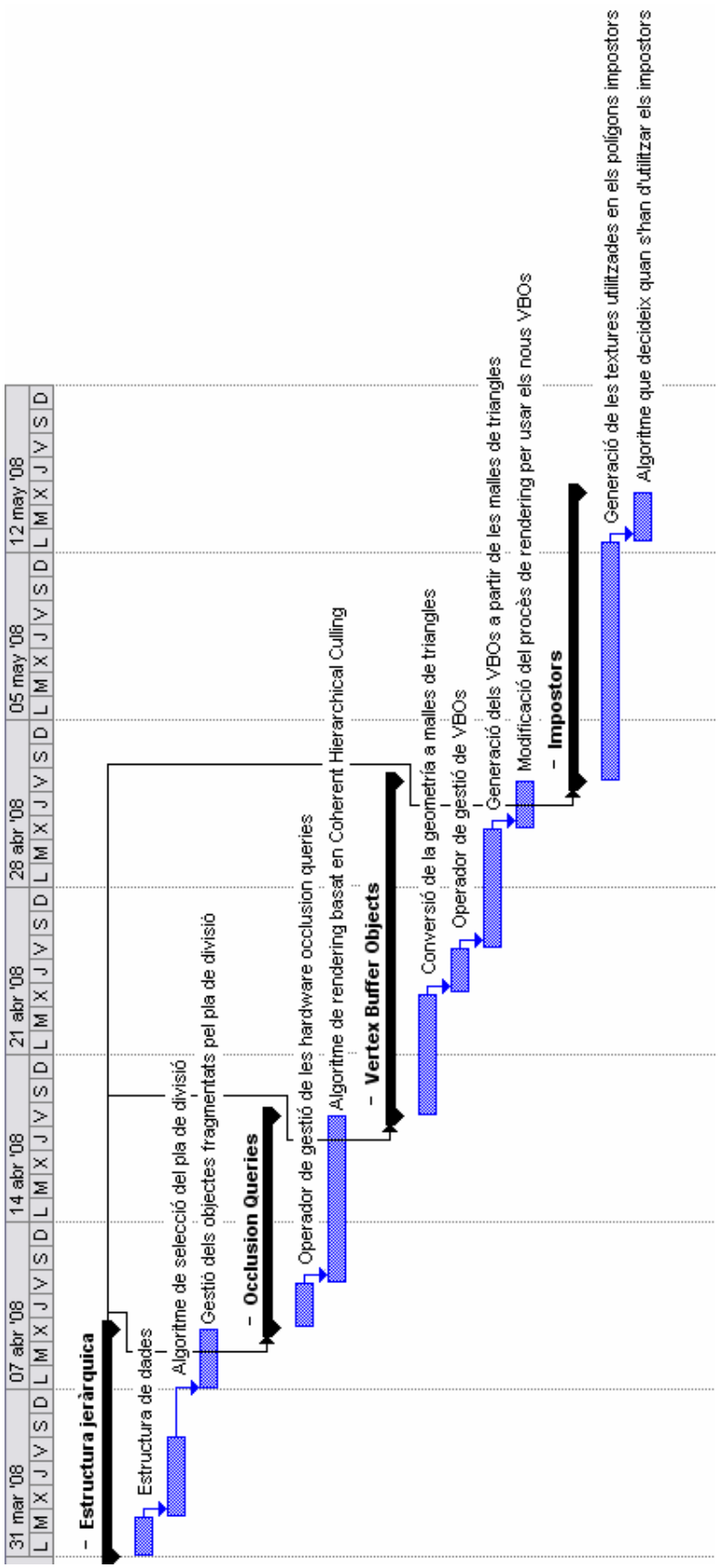


Figura 5.3.1: *Diagrama de Gantt de la implementació de les optimitzacions seleccionades.*

6 DESENVOLUPAMENT TÈCNIC

En aquest capítol es dona una descripció dels detalls tècnics relacionats amb el desenvolupament de les optimitzacions que s'han seleccionat. S'indiquen els principals canvis que s'han fet en el disseny d'*Alice*, així com els principals algoritmes utilitzats. D'aquesta manera, en cadascuna de les següents seccions es tracta una de les quatre millores implementades.

6.1 Estructura jeràrquica de divisió de l'espai

Aquest apartat tracta sobre el desenvolupament d'una estructura jeràrquica de divisió de l'espai amb les característiques indicades a l'anàlisi d'alternatives (descrites en el punt 4.2.2). S'expliquen les principals decisions de disseny que s'han pres, així com l'algoritme utilitzat per a seleccionar el pla de tall a cada pas de divisió.

6.1.1 Principals canvis en el disseny de l'aplicació

La nova estructura jeràrquica de divisió de l'espai (*K-d tree*) està basada en un arbre d'objectes de la nova classe *SpatialHierarchyNode*. Cadascuna de les instàncies d'aquesta classe representa a una regió de l'espai. Així doncs, cada node emmagatzema la caixa contenidora de la regió que representa (*BBox*) i la geometria del model continguda dins d'aquesta (nodes *BRep* de l'scene graph original). A més, per tal de gestionar la subdivisió recursiva de l'espai, també es guarda el pla que parteix cadascuna de les regions (pla coordenat ja que es tracta d'un *K-d tree*) i dos apuntadors cap a les noves subregions que es generen al dividir la regió original.

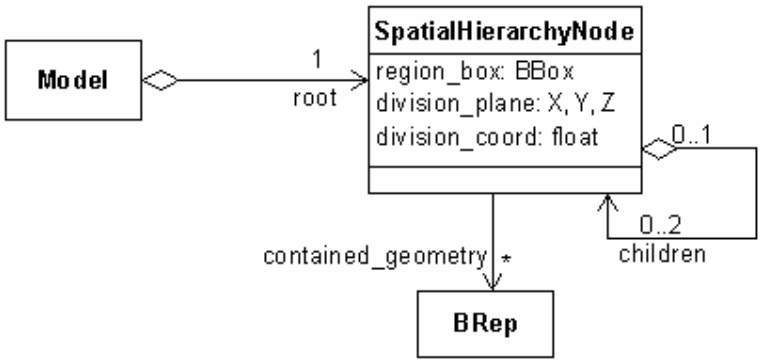


Fig. 6.1.1.1: *Diagrama d'estructura estàtica de l'estructura jeràrquica de divisió de l'espai.*

6.1.1.1 Construcció de l'estructura jeràrquica

El procés de construcció del *K-d tree* es gestiona des d'un nou operador anomenat *SpatialHierarchyBuilder*. Aquest, comença obtenint una llista amb tots els nodes de tipus *BRep* del model carregat. Això ho fa a partir d'un recorregut a través de la jerarquia de visualització de l'*scene graph*. A continuació, es crea el node arrel de l'estructura jeràrquica. Aquest conté tota la geometria del model (llista de *BReps* obtinguda) i ocupa una regió de l'espai corresponent a la caixa contenidora d'aquesta geometria.

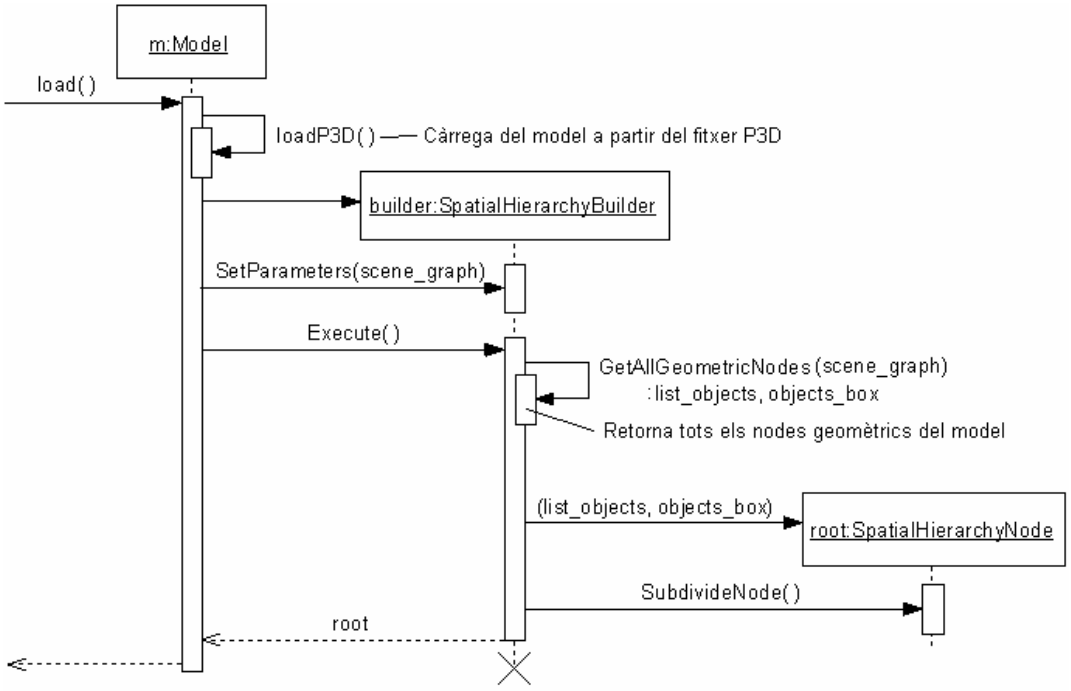


Fig. 6.1.1.1.1: *Diagrama de seqüència del procés de construcció de l'estructura jeràrquica mitjançant el nou operador SpatialHierarchyBuilder.*

Tot seguit, a partir d'aquest primer node, s'inicia un procés recursiu que va subdividint de forma progressiva l'espai de l'escena. A cada pas, es pren una de les regions existents, es selecciona un pla de tall (segons l'algoritme explicat al punt 6.1.2), es distribueix la geometria de la regió entre els dos subespais resultants i finalment, es creen dos nous nodes corresponents a les subregions afectades. El procés continua fins que s'assoleix el criteri de fulla establert (hi ha menys d'un cert nombre de vèrtexs o s'arriba a la profunditat màxima de 20 nivells). El valor òptim

per al llindar de vèrtexs vindrà determinat pel desenvolupament dels *VBOs* i s'estima que es situarà al voltant dels 50.000 vèrtexs.

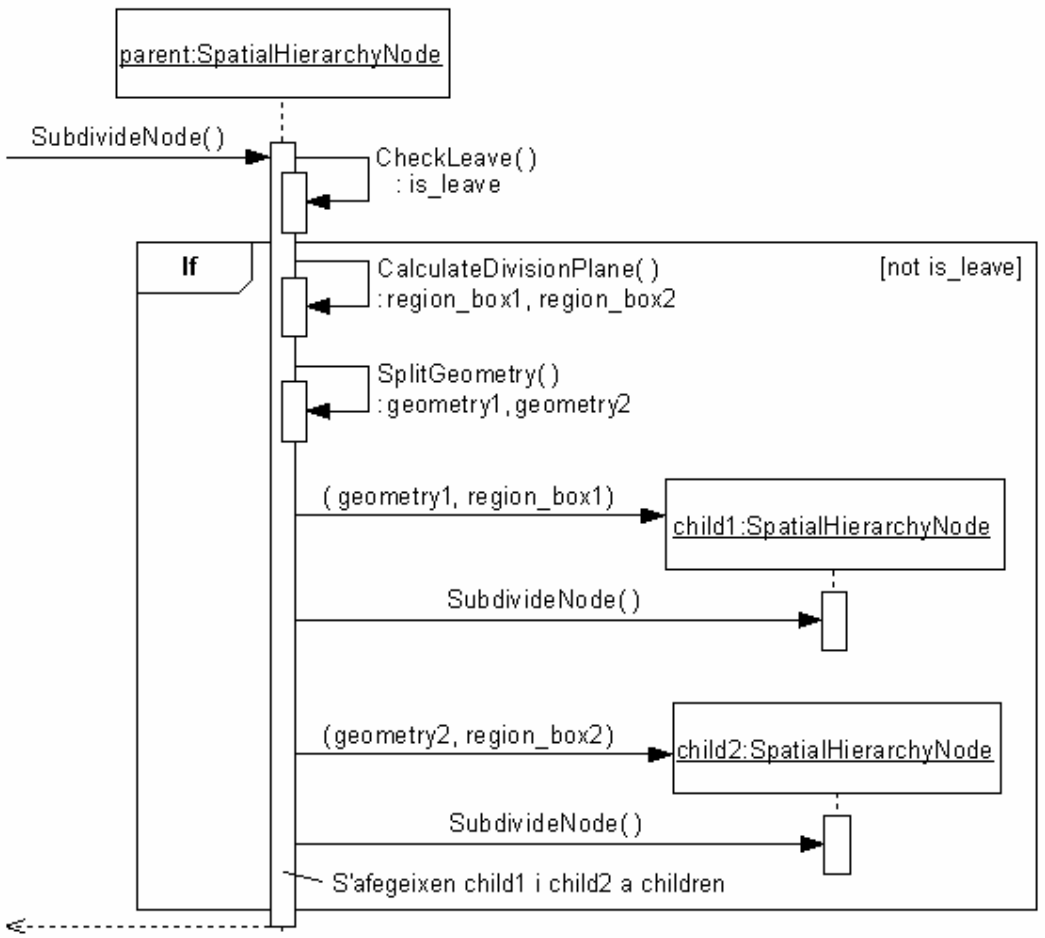


Fig. 6.1.1.1.2: *Diagrama de seqüència del procés de subdivisió de les regions de l'espai.*

Cal destacar que en un gran nombre d'ocasions hi haurà objectes (nodes *BRep*) intersecats pel pla de divisió. Tal i com s'ha comentat a l'anàlisi d'alternatives, això es resol a partir de tallar els políedres afectats. Així doncs, per cada objecte seccionat, es creen dos nous nodes *BRep*, de manera que cadascun conté les primitives que hi ha a un dels costats del pla. Les primitives intersecades pel pla es dupliquen i s'assignen als dos objectes.

6.1.1.2 Rendering mitjançant la nova estructura jeràrquica

La classe encarregada de gestionar el procés de *rendering* de l'aplicació és la ja existent *Render*. Com que *Alice* té la capacitat de visualitzar models amb independència del dispositiu de sortida utilitzat, l'objecte *Render* utilitza una classe auxiliar (*DisplayRender*) que s'encarrega de les tasques de *rendering* dels diferents tipus de dispositius (es té una subclasse per cadascun dels tipus que n'especialitza les particularitats). Així doncs, és el propi *DisplayRender* qui defineix les matrius *OpenGL* en funció dels paràmetres de la camera i invoca el rendering sobre el node arrel de l'estructura jeràrquica.

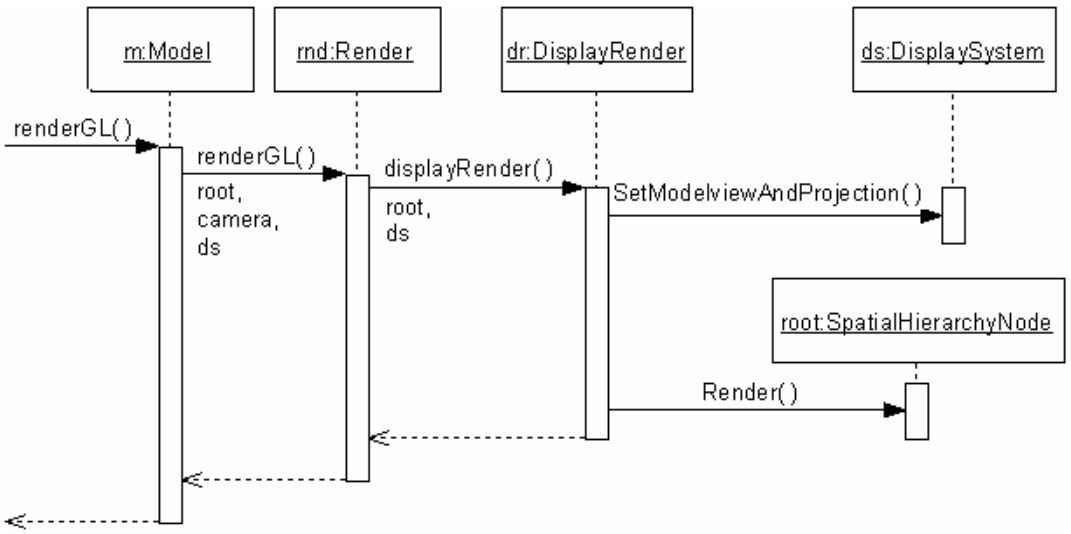


Fig. 6.1.1.2.1: *Diagrama de seqüència que mostra la relació entre les classes implicades en la gestió del procés de rendering de l'escena.*

El recorregut que es realitza a través de l'arbre s'efectua de forma descendent, fins arribar a les fulles d'aquest. Per cada fulla visitada, s'envia al *pipeline* d'*OpenGL* la geometria continguda a la regió corresponent. D'aquesta manera, s'aconsegueix independitzar el *rendering* de diferents zones de l'espai, cosa que serà útil per a posteriors optimitzacions.

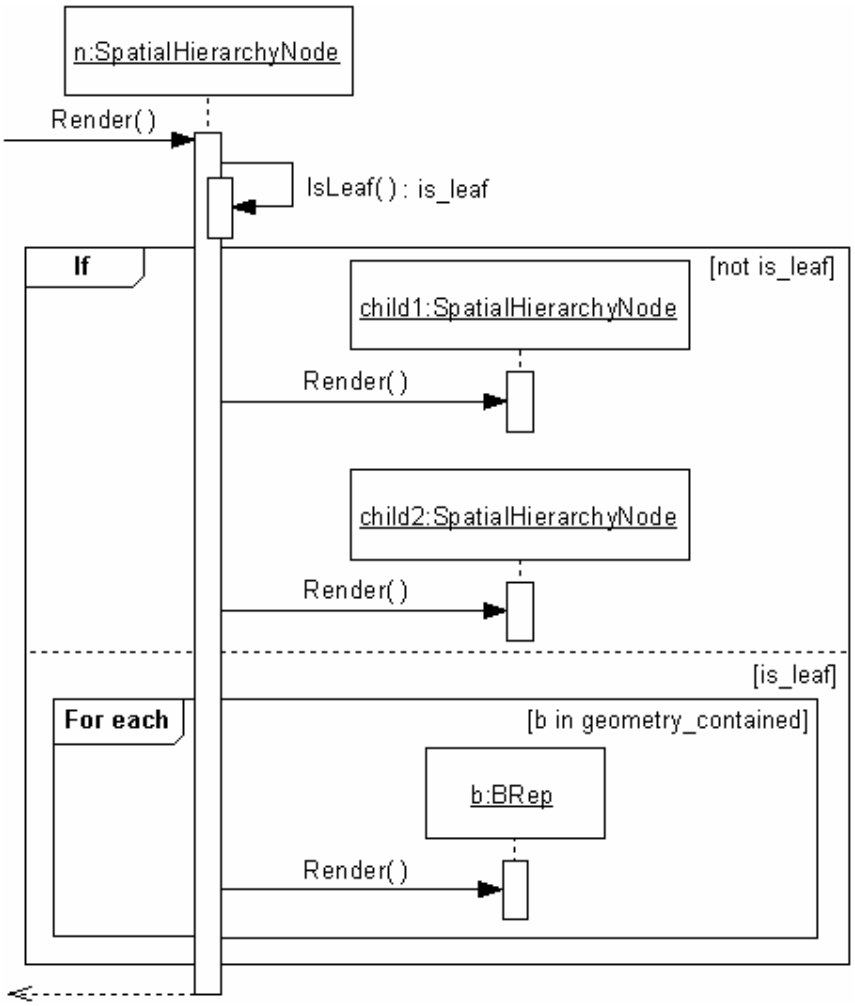


Fig. 6.1.1.2.2: *Diagrama de seqüència del recorregut realitzat sobre l'estructura jeràrquica per renderitzar l'escena.*

6.1.2 Algoritme de selecció del pla de tall

Tal i com s'ha pogut veure en el disseny del procés de construcció anterior, cal seleccionar quin és el pla de tall que s'utilitza per dividir les regions de cadascun dels nodes del *K-d tree*.

L'orientació d'aquests plans es disposa de manera que siguin perpendiculars a un dels tres eixos de coordenades. D'aquesta manera, es van alternant de forma repetida, les tres possibilitats existents en els diferents nivells de profunditat de la jerarquia: el primer nivell es dividirà amb un pla del tipus $x = k$, el segon nivell amb un pla del tipus $y = k$, el tercer amb un pla $z = k$, el quart novament amb un pla $x = k$, ...

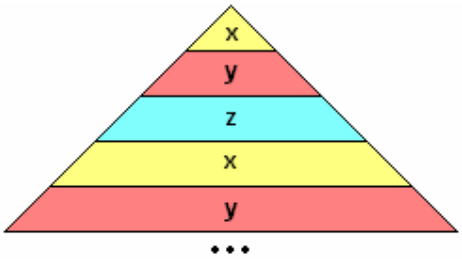


Fig. 6.1.2.1: *Distribució de l'orientació dels plans de tall entre els diferents nivells de profunditat.*

Així doncs, només resta per determinar per quin punt ha de passar el pla. Per fer-ho, es preveu utilitzar un algoritme que intenta tallar el mínim possible els objectes de la regió i que a la vegada els distribueix de forma el màxim d'equitativa possible entre les dues subregions. Consta dels següents passos:

1. *Generació de candidats*: Es prenen com a possibles candidats els plans de la caixa englobant de cadascun dels objectes continguts i que tenen l'orientació apropiada. Per cada candidat, es calcula quin nombre d'objectes intersecta i quantes primitives deixa a cada costat.

2. *Validació de candidats:* Només s'admeten aquells candidats que deixen a cada costat del pla de tall un nombre de primitives similar (dins un marge de tolerància d'un 10%). Tots els candidats que no satisfan aquesta condició són descartats.

3. Selecció del pla de tall: S'escull aquell candidat que intersecta amb el mínim nombre d'objectes. En cas d'empat, es selecciona aquell que reparteix de forma més equitativa les primitives de la regió.

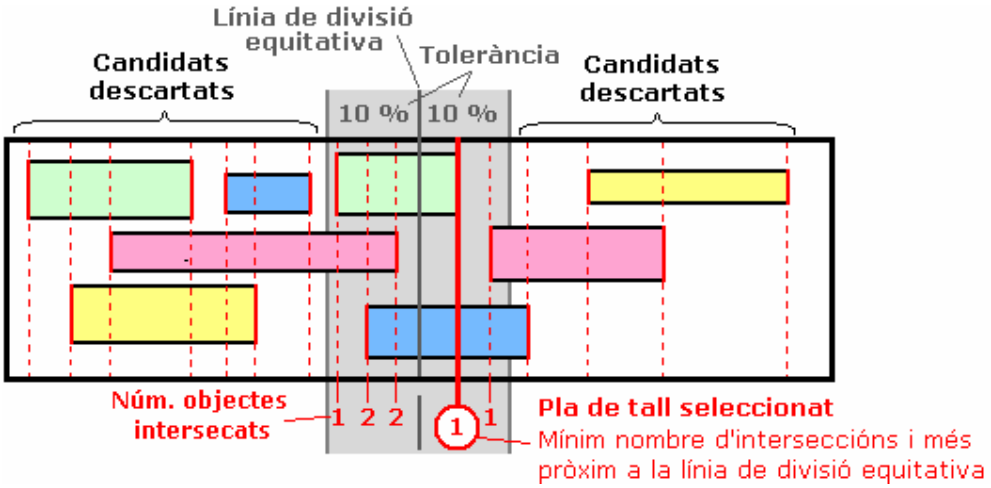


Fig. 6.1.2.2: *Selecció del pla de tall utilitzat per fer cadascuna de les divisions del K-d tree.*

6.2 Hardware occlusion queries

En aquest punt es comenten els aspectes més rellevants de la implementació d'un procés d'*occlusion culling* mitjançant *hardware occlusion queries* (veure el punt 4.3.1 de l'anàlisi d'alternatives). S'indiquen els canvis efectuats en el disseny de l'aplicació i es descriu l'algoritme *Coherent Hierarchical Culling*, encarregat de gestionar les *occlusion* queries per tal de simplificar el procés de *rendering.*

6.2.1 Principals canvis en el disseny de l'aplicació

Per tal de treballar còmodament amb les *hardware occlusion queries*, es decideix encapusular la seva gestió en una nova classe anomenada *OcclusionQuery*. Així doncs, cada instància d'*OcclusionQuery* representa una única consulta de visibilitat. Es permet llançar la seva execució (veure si una certa geometria que s'envia supera el test de profunditat), comprovar si els resultats es troben disponibles, i finalment recollir-los. Cal recordar que aquestes ofereixen el seu màxim rendiment quan s'executen de forma paral·lela.

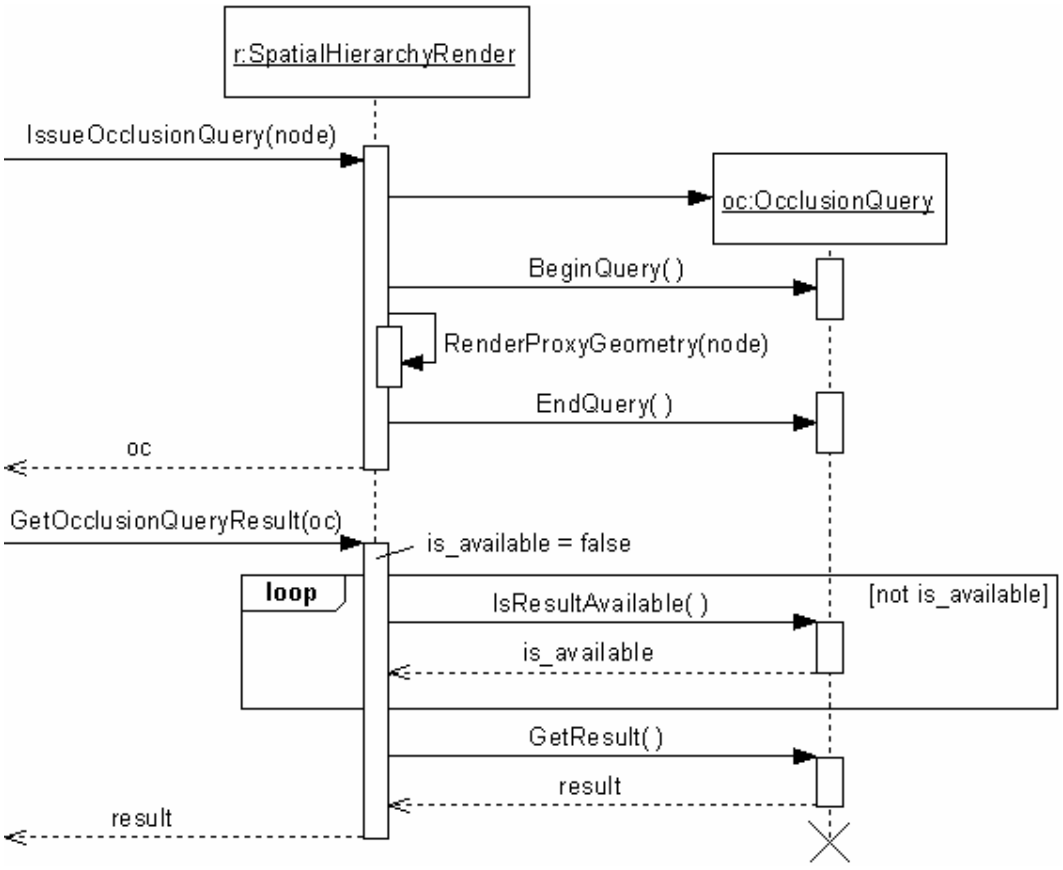


Fig. 6.2.1.1: *Diagrama de seqüència de l'execució típica d'una hardware occlusion query.*

Per altra banda, també es modifica el procés de visualització a través de l'estructura jeràrquica per tal que s'enviï a pintar la geometria de cada node segons l'algoritme *Coherent Hierarchical Culling* (explicat al punt 6.2.2). A més a més, per qüestions d'organització, es crea un nou operador (*SpatialHierarchyRender*) que permet albergar la implementació d'aquest algoritme.

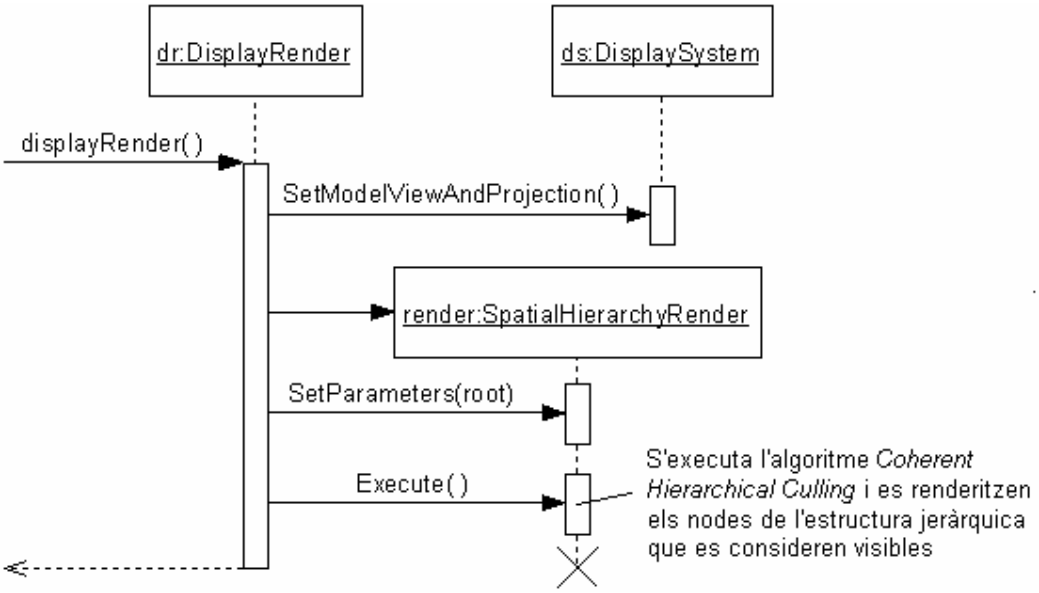


Fig. 6.2.1.2: *Diagrama de seqüència del procés de rendering a través del nou operador.*

Finalment, cal destacar que per requeriments del *Coherent Hierarchical Culling*, s'incorporen dos nous atributs als nodes de l'estructura jeràrquica. Concretament, s'afegeix un indicador de la visibilitat del node en el frame anterior i el valor de l'últim frame en que l'algoritme ha visitat el node.

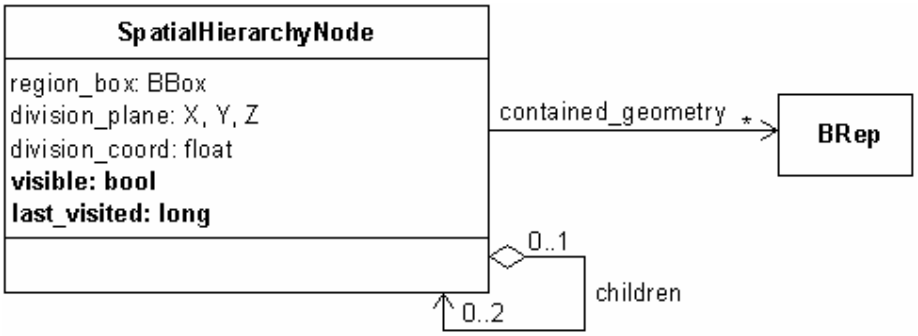


Fig. 6.2.1.3: *Nous atributs de la classe SpatialHierarchyNode.*

6.2.2 Algoritme *Coherent Hierarchical Culling*

Els nous càlculs *d'occlusion culling* es porten a terme mitjançant l'algoritme *Coherent Hierarchical* Culling, proposat a [9]. Aquest, utilitza conjuntament les *hardware occlusion queries* i una estructura jeràrquica de divisió de l'espai per a determinar quina geometria és visible i quina queda oculta. Addicionalment, s'ajuda de dues estructures de dades auxiliars:

- Una pila que conté els nodes de l'estructura jeràrquica que resten pendents de visitar. S'inicialitza amb el node arrel del *K-d tree*.
- Una cua amb les *hardware occlusion queries* enviades a la targeta gràfica i de les quals encara no s'ha recollit el resultat.

D'aquesta manera, l'algoritme es descomposa en dos fases perfectament diferenciades: el recorregut a través del *K-d tree* i el processament de les *hardware occlusion queries* que hagin finalitzat la seva execució. Aquestes dues fases, s'executen alternativament i de forma repetida fins que no finalitza el procés (la pila de nodes i la cua d'*occlusion queries* es troben buides).

En la primera, es pren el node del cim de la pila i es verifica si es troba dins el frustum de visió (*frustum culling*). En cas contrari, es descarta el node ja que amb total seguretat no serà visible. A continuació, es comprova si era visible en el frame anterior i si ha estat obert (era visible i no és una fulla). En cas que no s'hagi obert, es llança l'execució d'una nova occlusion query amb la geometria de la seva caixa englobant (*geometria proxy*) i s'afegeix a la cua. Finalment, en els casos en que el node era visible en el frame anterior es passa a processar el node. Això significa fer un recorregut a través dels seus fills (afegint-los a la pila en ordre *front-to-back*) o en el cas que sigui una fulla, a renderitzar-lo.

La segona fase recull el resultat de les *occlusion queries* llançades una vegada han completat el seu processament. En el cas que la pila no contingui cap node per a visitar, s'espera que la primera *query* de la cua acabi d'executar-se. El resultat obtingut consisteix en el nombre de pixels de la caixa englobant del node que superen el *z-test*. Si aquest valor és inferior a un cert llindar (cal determinar-lo experimentalment), aleshores el node es considera no visible i es descarta. En cas contrari, es processa el node i es marquen tots els nodes predecessors com a visibles.

Algoritme 1. Recorregut a través del K-d tree.

```
TraversalStack.Push(kDTree.Root);
while ( not TraversalStack.Empty() or
        not QueryQueue.Empty() )
{
   //-- PART 1: recorregut a través del K-d tree.
   if ( not TraversalStack.Empty() )
   {
      N = TraversalStack.Pop();
      if ( InsideViewFrustum(N) )
      {
         // identifica nodes visibles previament
         wasVisible = N.visible && (N.lastVisited == frameID -1);
         // identifica nodes visitats previament (oberts)
         opened = wasVisible && !IsLeaf(N);
         // reinicialitza els flags del node
         N.visible = false;
         N.lastVisited = frameID;
         // evita el test de nodes visitats previament
         if ( !opened )
         {
            oc = IssueOcclusionQuery(N);
            QueryQueue.Enqueue(oc, N);
         }
         // fa un recorregut pel node encara que sigui invisible
         if ( wasVisible )
            TraverseNode(N);
      }
   }

   //-- PART 2: processament de les occlusion queries completades.
   while ( not QueryQueue.Empty() and
          (ResultAvailable(QueryQueue.Front()) or
           TraversalStack.Empty()) )
   {
      <oc,N> = QueryQueue.Dequeue();
      // espera si el resultat no està disponible
      visiblePixels = GetOcclussionQueryResult(oc);
      N.visible = visiblePixels > VisibilityThreshold;
      if ( N.visible )
      {
         TraverseNode(N);
         PullUpVisibility(N);
      }
   }
}
```

Algoritme 1a. Recorregut d'un únic node.

```
TraverseNode(N)
{
   if ( IsLeaf(N) )  Render(N);
   else              TraversalStack.PushChildren(N);
}
```

Fig. 6.2.2.1: *Pseudo-codi de l'algoritme Coherent Hierarchical Culling.*

6.3 Vertex Buffer Objects (VBOs)

El contingut d'aquest punt fa referència a la implantació de *Vertex Buffer Objects* (*VBOs*) per tal de processar de forma més eficient la geometria que s'envia al *pipeline* gràfic (punt 4.1.1 de l'anàlisi d'alternatives). Es mostren les solucions de disseny que s'han escollit per aquest desenvolupament i finalment, s'expliquen els principals algoritmes involucrats.

6.3.1 Principals canvis en el disseny de l'aplicació

La utilització de *VBOs* a *Alice* suposa modificar el disseny de l'aplicació en una sèrie de punts ben diferenciats. Cal convertir la geometria del model en malles de triangles, gestionar la creació dels VBOs, incorporar-los dins l'estructura jeràrquica i fer un tractament especial de les coordenades de textura. Així doncs, en les pròximes seccions es fa referència a cadascun d'aquests punts.

6.3.1.1 Conversió de la geometria en malles de triangles

Per poder treballar visualitzar la geometria del model mitjançant *VBOs* fa falta tenir-la expressada en forma de malla de triangles. Com que actualment *Alice* enregistra els objectes com una llista de primitives, cal efectuar una transformació prèvia. A més, també es necessita una nova estructura de dades per a gestionar les noves malles que es generin. D'aquesta manera, es creen dues noves classes, que s'anomenen *SpatialHierarchyMesh* i *MeshVertex*, i que permeten representar respectivament aquestes noves malles i els vèrtexs que les composen.

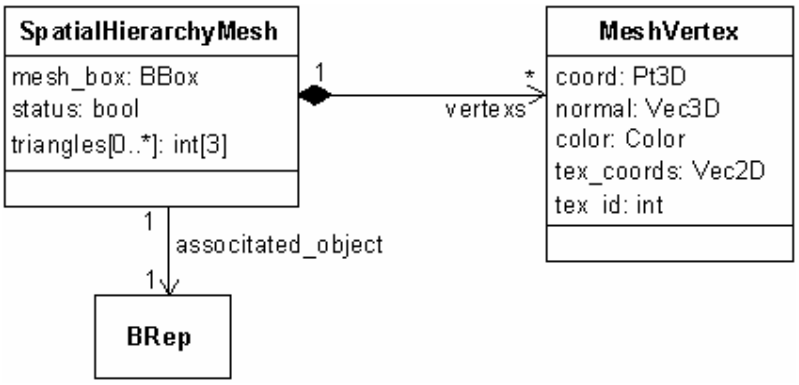


Fig. 6.3.1.1.1: *Estructura de dades per a gestionar les noves malles de triangles.*

La classe *SpatialHierarchyMesh* bàsicament consisteix en un vector de vèrtexs (instàncies de *MeshVertex*) i en una llista de triangles, on els triangles es representen mitjançant tres índexs que referencien al vector de vèrtexs. També s'emmagatzema el node *BRep* a partir del qual es genera la malla, la caixa contenidora i un *flag* que indica si el seu estat és correcte. En relació als vèrtexs, tot i que ja existia la classe *Vertex* que guardava aquesta informació, es fa necessari crear un nou tipus d'objecte per a incloure-hi les dades relatives a la texturació (fins al moment es guardaven a nivell de primitiva). En el punt 6.3.1.4 s'explica amb detall quin tractament es fa per al cas de models texturats.

Així doncs, el nou procés de conversió partirà de tots els nodes *BRep* del model i obtindrà una malla de triangles per a cadascun (l'algoritme utilitzat es comenta al punt 6.3.2.1). Cal tenir present que aquest procediment s'ubica a l'operador *SpatialHierarchyBuilder*, de manera que es modifica la construcció de l'estructura jeràrquica per a que es substitueixin els nodes *BRep*, per malles de triangles. El concepte del procés és el mateix, però s'adapta per a que treballi amb la nova estructura de dades de la geometria.

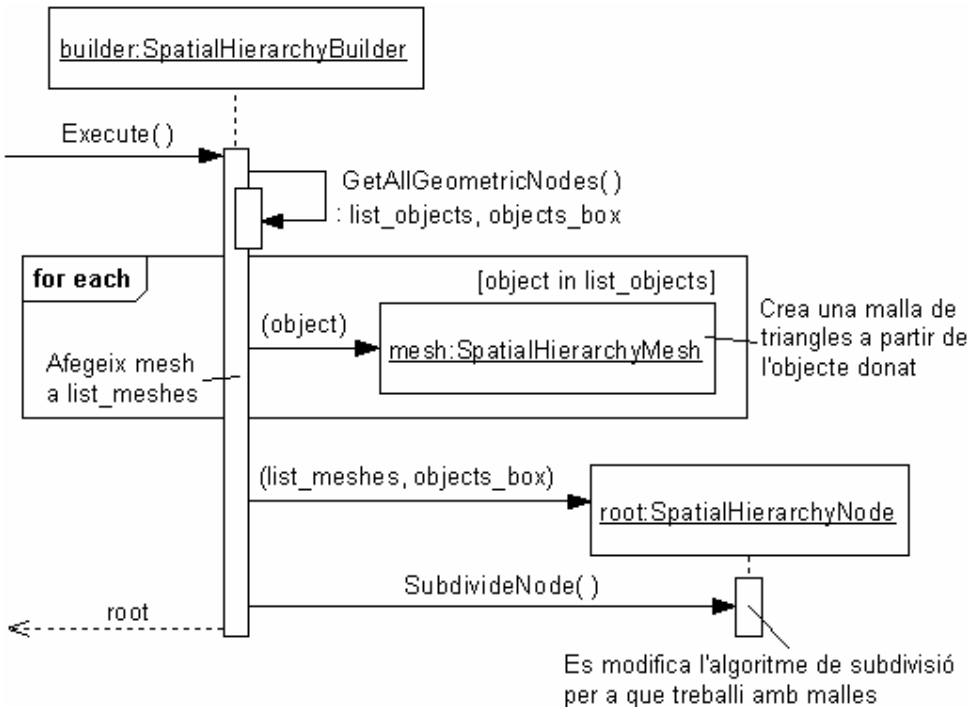


Fig. 6.3.1.1.2: *Diagrama de seqüència de la conversió de la geometria del mode en malles de triangles, durant la creació de l'estructura jeràrquica.*

Finalment, un dels principals inconvenients d'aquest procés es troba en els precàlculs d'il·luminació (es computa per software el color de cada vèrtex). Fins al moment, aquests es desenvolupaven en un *thread* auxiliar per tal de reduir el temps de càrrega del model (es podia començar a visualitzar l'escena abans que es completessin). Això ha de deixar de ser així degut al procés de conversió de la geometria. Aquest té com a precondició que el color de tots vèrtex ja estigui calculat. Altrament, el model es veuria parcialment il·luminat. D'aquesta manera, per poder garantir aquest requisit, es mouen tots els càlculs d'il·luminació al flux d'execució principal.

6.3.1.2 Gestió dels Vertexs Buffer Objects (VBOs)

Amb la finalitat de facilitar la gestió dels *VBOs* (generació, rendering, destrucció, ...), es crea un nou tipus d'objecte que s'encarrega de totes aquestes tasques. La nova classe s'anomena *VertexBufferObject* i permet administrar els *buffers* d'*OpenGL* necessaris per a renderitzar eficientment una malla de triangles.

Addicionalment, s'ofereix la possibilitat de modificar les característiques dels *buffers* que s'utilitzin per emmagatzemar les dades dels vèrtexs. En particular, es dona l'opció d'especificar quina informació es guarda (posició, normal, color i coordenades de textura), amb quin tipus de dades es representen aquestes components i com s'organitzen les dades (un únic *buffer*, un *buffer* diferent per cada per component, dades intercalades, ...). D'aquesta manera, en els punts 7 i 8 es fan proves que determinen quina és la configuració de tots aquests paràmetres que millors resultats dona.

6.3.1.3 Incorporació dels VBOs a l'estructura jeràrquica

Tal i com s'ha explicat a l'anàlisi d'alternatives, es pretén fer servir el *K-d tree* per a distribuir la geometria del model entre els diferents *VBOs* que es construeixin. La idea consisteix en associar un *VBO* per cada node fulla de l'estructura jeràrquica, de manera que aquest contingui totes les malles de triangles contingudes a la regió del node. Com que les regions de tots els nodes fulla constitueixen una partició disjunta de l'espai de l'escena, així s'assegura que tota la geometria del model quedi perfectament repartida.

D'aquesta manera, es torna a canviar l'estructura estàtica de la classe *SpatialHierarchyNode*, per afegir-hi una referència a l'objecte *VertexBufferObject*

corresponent al *VBO* del node. L'aspecte de la classe un cop s'ha modificat és el següent:

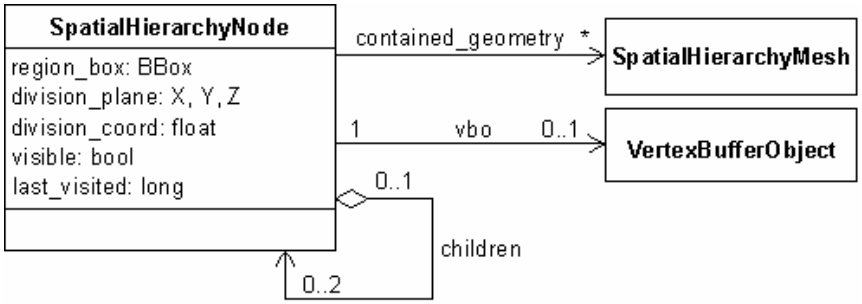


Fig. 6.3.1.3.1: *Representació UML de la nova versió de la classe SpatialHierarchyNode.*

Pel que fa a la construcció de les instàncies dels *VBOs*, aquesta s'inicia des de l'operador *SpatialHierarchyBuilder*, un cop completada la fase de subdivisió. Es fa un recorregut per tots els nodes del *K-d tree*, de manera que per cada fulla que es visita, es construeix una malla que fusiona totes la es malles que conté i es construeix un *VBO* amb aquesta malla fusió.

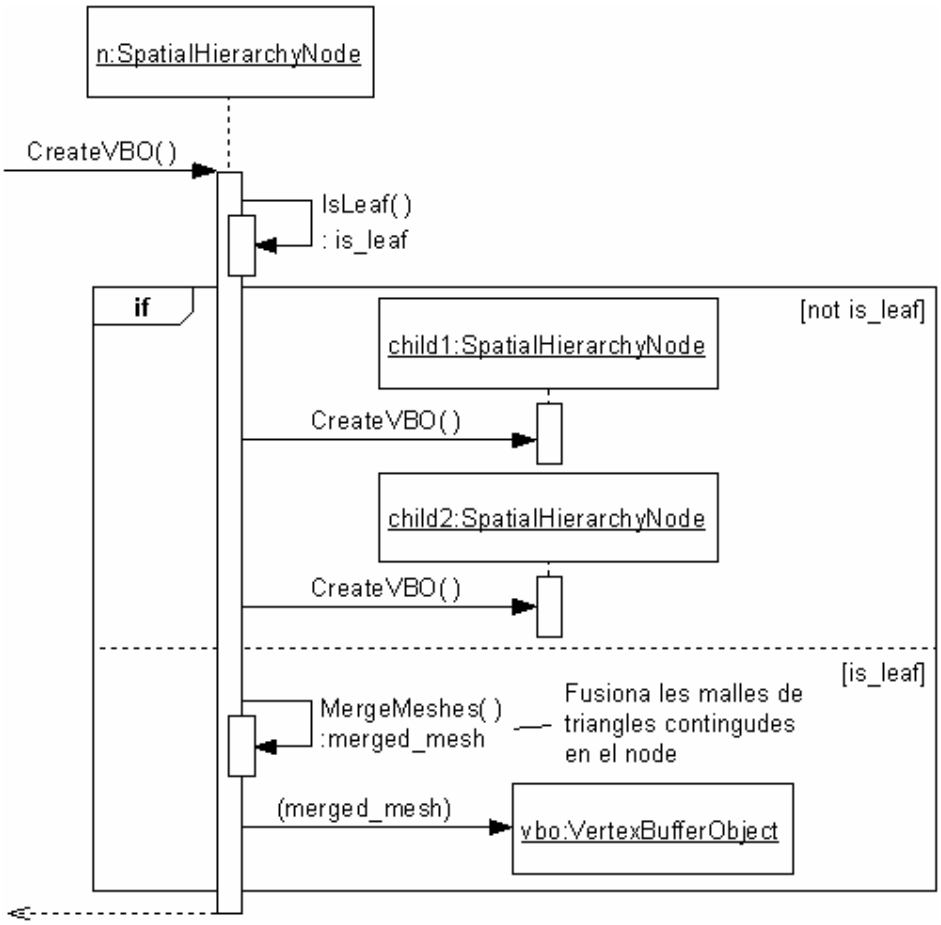


Fig. 6.3.1.3.2: *Diagrama de seqüència del procés de construcció dels VBOs.*

En relació amb el procés de *rendering*, cada cop que es decideix enviar a pintar un node de l'estructura jeràrquica, es comprova si aquest disposa de *VBO*. En cas afirmatiu, es procedeix a utilitzar-lo per a fer el *rendering* corresponent. En cas contrari, s'envia a pintar la geometria a través del mode immediat d'*OpenGL*.

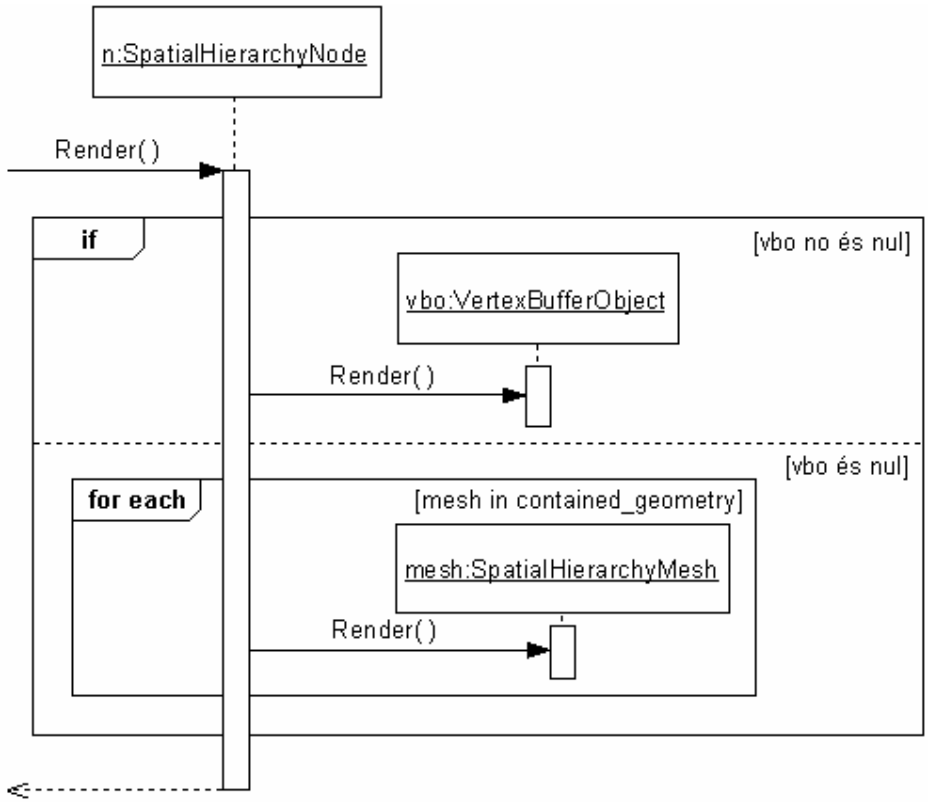


Fig. 6.3.1.3.3: *Diagrama de seqüència del procés de rendering mitjançant VBOs.*

6.3.1.4 Tractament de les coordenades de textura

La implantació de *VBOs* requereix fer una gestió especial de les coordenades de textura per a poder visualitzar correctament els models que les utilitzen. En primer lloc, cal disposar d'aquestes coordenades a nivell de vèrtex, juntament amb l'identificador de la textura a la que fan referència). Fins al moment, s'estaven guardant a nivell de primitiva, però això no pot continuar essent així ja que al passar a treballar amb malles de triangles, tota aquesta informació es mou fins als vèrtexs (objectes *MeshVertex*).

En aquest punt, s'ha de posar èmfasi en el tractament que es dona als vèrtexs compartits per diverses primitives durant la generació de les malles. En el cas que tinguin coordenades de textura amb valors diferents o si referencien a textures

diferents, caldrà replicar-los (encara que es trobin a la mateixa posició). Altrament, seria impossible texturar correctament tots els triangles que l'utilitzen.

Per altra banda, també s'ha de tenir molt present la manera com es renderitzen els *VBOs*: de cop, en un únic pas. En aquesta situació, l'estat d'*OpenGL* s'ha de mantenir constant per al processament de tota la geometria d'un *VBO*. Això significa que només es pot tenir activa una única textura per a pintar tots els triangles del *VBO*, cosa que suposa un problema important si se n'utilitzen varies.

Una solució consistiria en fer el *rendering* en diferents etapes (tantes com textures s'utilitzin), de manera que a cadascuna s'enviessin a pintar únicament els triangles texturats amb una de les textures. Es desestima aquesta alternativa ja que es complicaria molt la gestió dels *VBOs* (caldria classificar els triangles segons les textures). A més, quan el nombre de textures fos elevat, es perdrien gran part dels guanys de rendiment que aporten els *VBOs*.

D'aquesta manera, es decideix resoldre aquest problema mitjançant la creació d'un atlas de textura [25]. La idea consisteix en generar una nova textura d'*OpenGL* (atlas), de grans dimensions, tal que el seu contingut consisteixi amb totes les textures del model, disposades una al costat de l'altra.

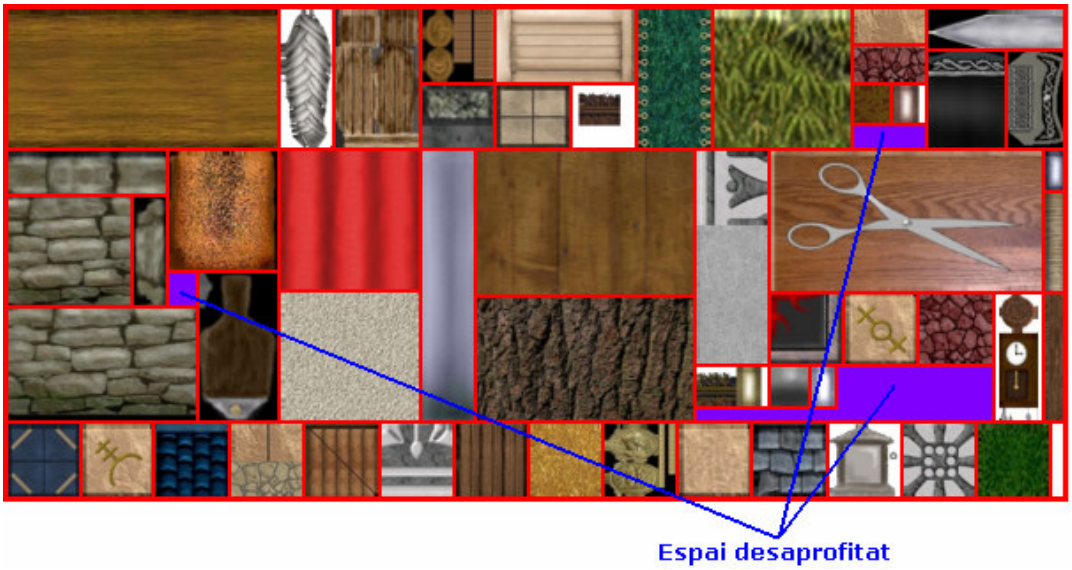


Fig. 6.3.1.4.1: *Exemple d'atlas de textura.*

A continuació, es recalculen totes les coordenades de textura dels model per a que facin referència al punt homòleg sobre l'atlas. Així, s'aconsegueix prescindir de les textures originals ja que tota la informació que contenen es troba dins l'atlas.

Finalment, es renderitza tot el contingut del *VBO* posant l'atlas generat com a textura activa.

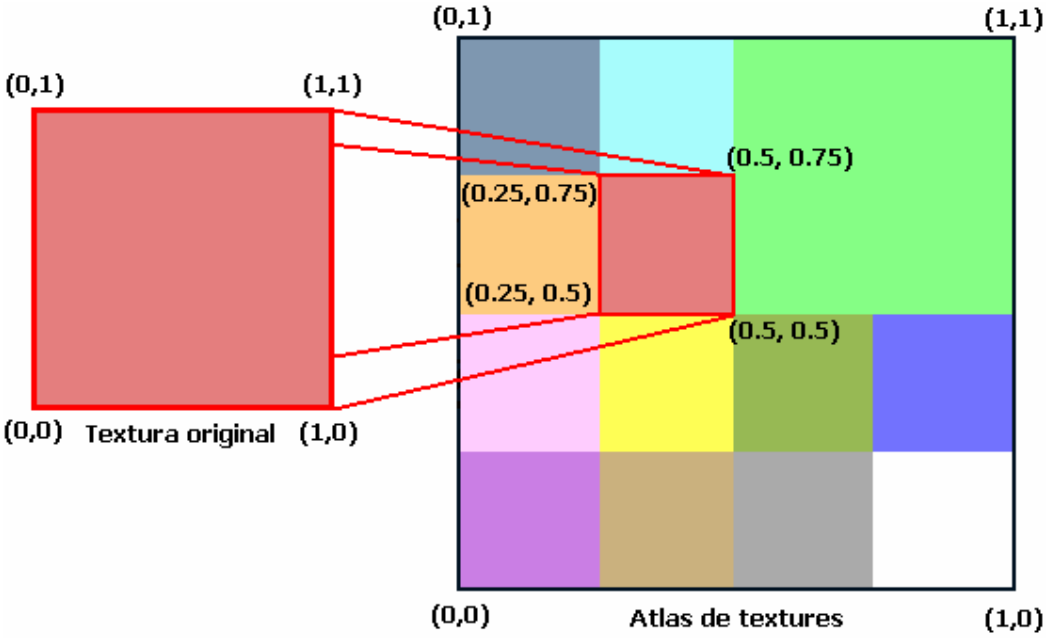


Fig. 6.3.1.4.2: *Recàlcul de les coordenades de textura per tal que facin referència a l'atlas.*

Tot i això, cal tenir present que l'atlas, al tractar-se d'un textura *OpenGL*, ha de tenir forma rectangular. Això significa que cal distribuir les textures (també rectangulars) de manera que es minimitzi l'espai desaprofitat. Com a conseqüència, el tamany final de l'atlas i la memòria necessària per a emmagatzemar-lo també serà mínima. L'algoritme descrit al punt 6.3.2.2 s'encarrega d'aquesta qüestió.

També s'ha de tenir en compte el cas dels models que usen textures, però que a la vegada tenen triangles sense texturar. Per resoldre aquest tipus de situacions, es planteja incloure un pixel transparent a l'atlas i fer que les coordenades de textura de les primitives afectades hi apuntin. D'aquesta manera, quan es rasteritzi la geometria sense texturar, s'utilitzarà el color dels vèrtexs que la defineixen. En qualsevol cas però, per tal que funcioni correctament el mecanisme de transparències, és necessari que el mode de texturació que ve definit per la variable d'*OpenGL GL_TEXTURE_ENV_MODE* tingui com a valor *GL_MODULATE*.

Un altra problemàtica relacionada amb la creació de l'atlas de textura es troba amb els modes d'adreçament que permeten tractar coordenades de textura fora del rang [0,1]. *Alice* considera la utilització del mode de *wrap,* el qual descarta la part entera de les coordenades i utilitza només la component fraccionaria per adreçar la

textura. Això produeix l'efecte visual d'una repetició continuada de la textura (veure figura 6.3.1.4.3), molt útil per a modelar grans superfícies amb textures petites (ex. rajols d'una paret, camp de gespa, ...).

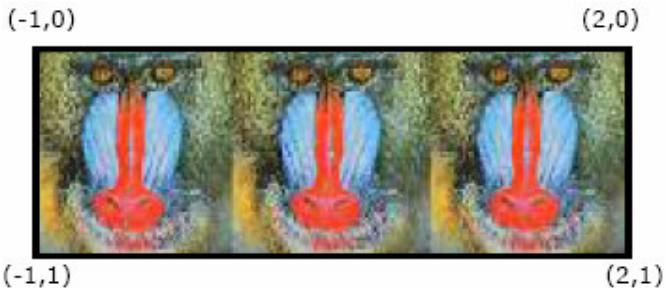


Fig. 6.3.1.4.3: *Quad texturat amb el mode d'adreçament wrap.*

Tot i això, les coordenades que mapejen el posicionament d'una textura sobre l'atlas es troben en un subconjunt estricte de [0,1]. Per tant, en el cas d'una textura situada al centre de l'atlas, els modes d'adreçament no s'apliquen. En el seu defecte, s'accedeix a les textures veïnes en l'atlas, produint un resultat no desitjat.

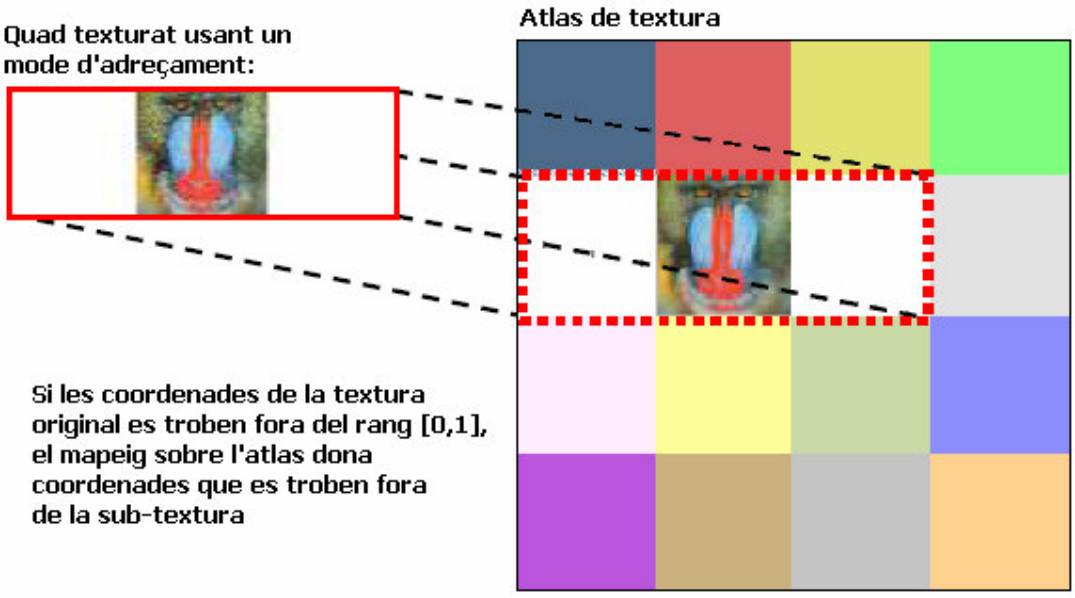


Fig. 6.3.1.4.4: *Problemàtica associada als modes de wrapping i a l'atlas de textura.*

Una possible forma d'esquivar aquesta situació consisteix en replicar la mateixa textura múltiples cops a l'atlas. D'aquesta manera, si una textura es repeteix fins a *n* cops, llavors es copia *n* vegades a l'atlas. Aquesta tècnica malgasta grans

quantitats de memòria de textura, especialment quan s'usen simultàniament els modes d'adreçament sobre els eixos *u* i *v*. A més, es complica notablement l'algoritme de construcció de l'atlas. Per tant, no es considera que aquesta sigui una alternativa vàlida per a resoldre aquest problema.

Un altra possibilitat consisteix en descompondre les primitives que tinguin vèrtexs amb coordenades de textura fora del rang [0,1]. Això es fa de manera que tots els triangles generats tinguin coordenades de textura que no requereixin fer ús dels modes d'adreçament. Augmenta la complexitat del model original (es té un major nombre de triangles), però no requereix més espai per a l'atlas (memòria de textura).

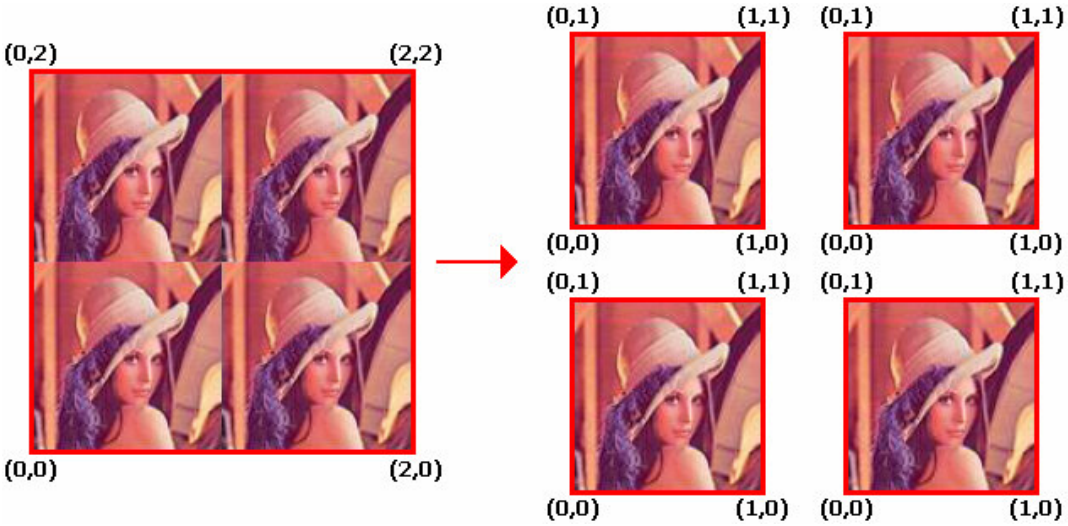


Fig. 6.3.1.4.5: *Subdivisió d'un quadrilàter per a evitar utilitzar els modes de d'adreçament.*

Així doncs, s'ha decidit implementar un algoritme que efectua una triangulació d'aquestes característiques. Es llança just després de convertir la geometria del model a malles de triangles, però abans de construir l'atlas.

Finalment, per a gestionar la construcció de l'atlas i el nou mapeig de les coordenades de textura de cada vèrtex, es crea un nou operador anomenat *TextureAtlas*. Concretament, s'encarrega de determinar quines textures utilitza el model, d'encapsular l'algoritme que les distribueix, de generar la texura *OpenGL* de l'atlas i de proporcionar un mecanisme per convertir fàcilment les coordenades de textura (s'utilitza un mapa que guarda on es posiciona cada textura original sobre l'atlas). S'invoca des de l'operador *SpatialHierachyBuilder*, just després de generar les malles de triangles.

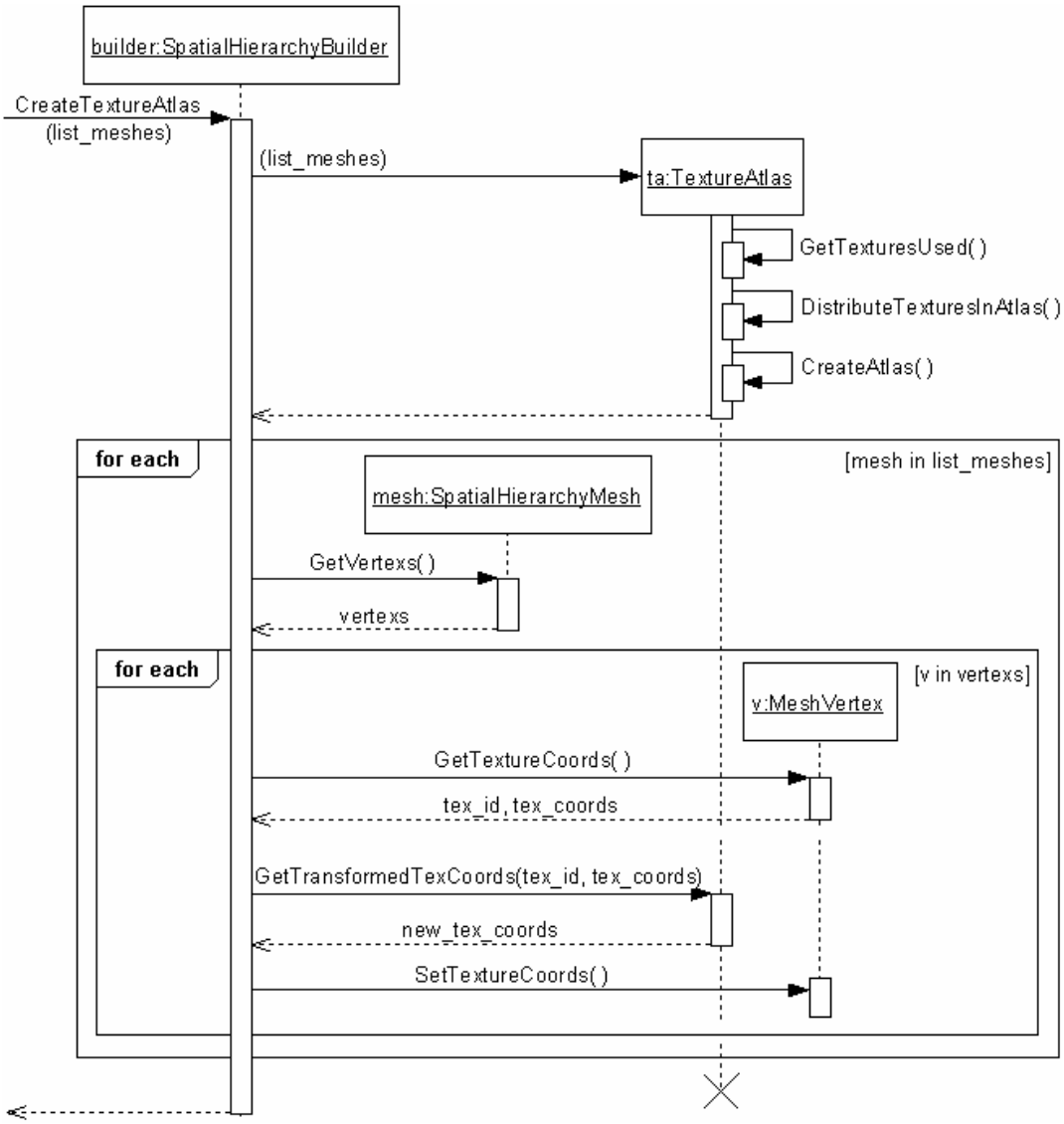


Fig. 6.3.1.4.3: *Diagrama de seqüència de la construcció de l'atles de textura.*

6.3.2 Principals algoritmes

6.3.2.1 Algoritme de generació de la malla de triangles

La idea bàsica d'aquest algoritme consisteix en fer un recorregut a través de la llista de primitives d'un objecte per tal de generar la malla de triangles corresponent. D'aquesta manera, per cada primitiva que es visiti s'haurà de fer el següent:

1. En el cas que la primitiva no sigui un triangle, caldrà triangular-la. Com que només es considera la utilització de polígons convexos, aquesta triangulació es farà prenent un vèrtex base (comú per a tots els triangles que es formin) i les parelles de vèrtexs corresponents a totes les arestes de la primitiva que no incideixen sobre el vèrtex base (veure figura 6.3.2.1.1). Addicionalment, també s'han de convertir a triangles les primitives particulars d'*OpenGL* (*strips* i *fans*), utilitzades per a reduir el nombre de vèrtexs processats.

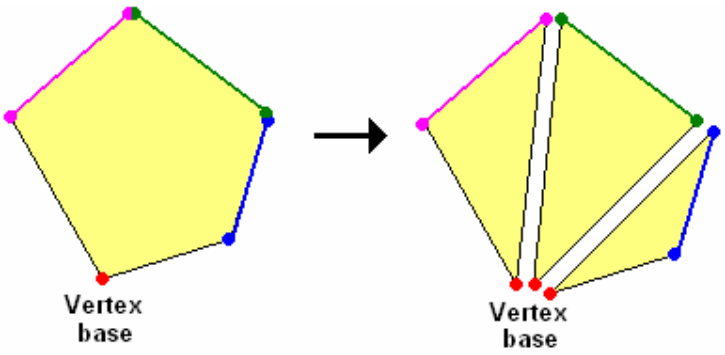


Fig. 6.3.2.1.1: *Algoritme de triangulació d'un polígon convex.*

2. S'afegeixen els vèrtexs de la primitiva al vector de vèrtexs de la malla. Abans de realitzar la inserció, es comprova que el vèrtex no es trobi ja dins el vector (vèrtex compartit en triangles contigus). Per fer-ho, cal comparar la seva posició i els seus atributs (ex. coordenades de textura) amb els de tots els altres vèrtexs del vector. Per tant, el cost d'aquest pas és relativament elevat ($O(n^2)$ respecte el nombre de vèrtexs). Tot i això, donat que aquest algoritme només s'executa al carregar el model, no suposa un punt crític per al rendiment de l'aplicació.

3. S'inserten al vector de triangles els índexs del vector de vèrtexs que corresponen als vèrtexs de cadascun dels triangles en què es descompon la primitiva.

6.3.2.2 Algoritme de construcció de l'atlas de textures

El problema de la distribució de les textures del model sobre la superfície de l'atlas es pot reduir al conegut problema de la motxilla (*knapsack*) [26], extès a l'espai 2D. Com que es tracta d'un problema *NP-complet*, no es disposa de cap solució en temps polinomial (respecte al nombre de textures) per a resoldre'l.

De totes maneres, donat que es tracta d'una matèria molt estudiada, existeix un gran ventall d'algoritmes que l'afronten [27]. Acostumen a ser capaços de proporcionar una solució en un temps acceptable, però a la vegada també són complexes i difícils de comprendre. Per aquest motiu, juntament amb que s'espera que el nombre de textures tampoc sigui excessivament gran, es decideix adoptar una solució *ad-hoc*. Aquesta, troba un equilibri entre la simplicitat, el cost computacional i el tamany de l'atlas, que la fa perfectament vàlida per als propòsits de l'aplicació.

Així doncs, l'algoritme proposat consisteix en fer un recorregut per totes les textures del model i anar afegint-les una a una dins l'atlas. Es parteix d'un atlas de dimensions 0x0 i progressivament es va incrementant el seu tamany per tal de donar cabuda a les textures que s'hi insereixen.

La posició on es col·loquen les textures a cada pas es selecciona d'una llista de candidats, de manera que es minimitzi l'augment de l'atlas i que no hi hagi solapament amb les textures ja situades (figura 6.3.2.2.1). Pel que fa a la llista de candidats, aquesta conté el conjunt de possibles punts on posar la textura (els punts en representen la cantonada inferior esquerra). S'inicialitza amb el punt (0,0) i per cada textura que s'afegeix, es creen dos nous candidats (corresponents a les cantonades inferior dreta i superior esquerra), tal i com mostra la figura 6.3.2.2.2.

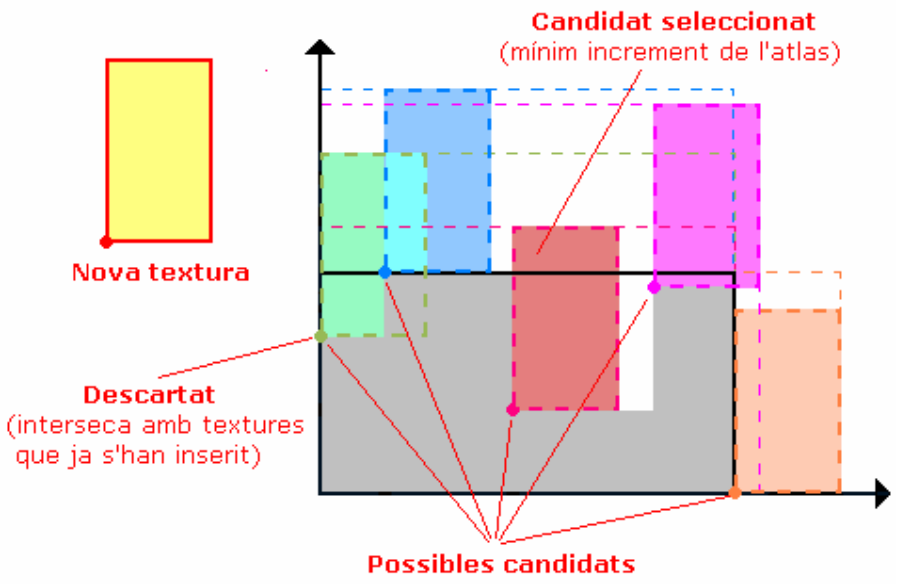


Fig. 6.3.2.2.1: *Selecció del candidat que minimitza l'increment de la mida de l'atlas i que no es solapa amb cap de les textures ja inserides.*

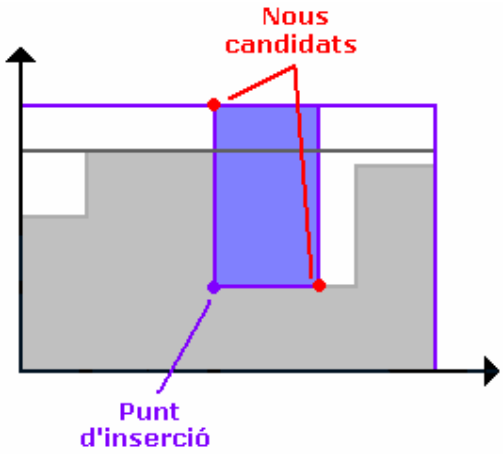


Fig. 6.3.2.2.2: *Nous candidats creats al afegir una textura dins l'atlas.*

També cal destacar que les textures es recorren en ordre, segons l'àrea que tenen aquestes (de gran a petita). Això es fa pensant en el fet que les textures grans són més difícils de col·locar que les petites. D'aquesta manera, s'adopta l'estratègia de situar les grans en primera instància i després, tapar els forats restants amb les més petites.

6.4 Impostors

L'últim desenvolupament abordat per aquesta tesi consisteix en la representació de la visualització de la geometria llunyana mitjançant impostors (veure apartat 4.4.2). Com en els punts anteriors, es descriuen els principals canvis en el disseny de l'aplicació i els algoritmes introduïts més destacables.

6.4.1 Principals canvis en el disseny de l'aplicació

El tipus d'impostor seleccionat a l'anàlisi d'alternatives consisteix en una sèrie de polígons texturats associats a un dels nodes de l'estructura jeràrquica. Concretament, cada impostor està format per 6 polígons, els quals recobreixen les parets de la caixa contenidora que representa la regió de l'espai definida pel node. Així doncs, es crea una nova classe, *SpatialHierarchyImpostor*, on cada instància d'aquesta enregistra les textures i el posicionament dels polígons d'un impostor. També es modifica de nou la classe *SpatialHierarchyNode* per tal que l'estructura jeràrquica pugui inclogui el vincle amb l'objecte impostor corresponent. El diagrama de classes de la figura 6.4.1.1 mostra com queda l'estructura de dades resultant:

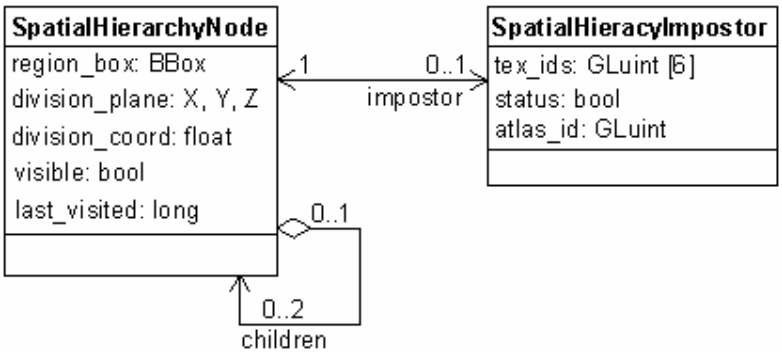


Fig. 6.4.1.1: *Diagrama de classes de l'estructura jeràrquica un cop s'han inclòs els impostors.*

El procés de generació dels impostors es situa com a últim pas de la construcció de l'estructura jeràrquica. S'inicia des de l'operador *SpatialHierarchyBuilder* i consisteix en un recorregut a través de tots els nodes del *K-d tree*. Per cada node visitat, s'obté la caixa contenidora de la regió corresponent. Tot seguit, es posiciona una càmera per cadascuna de les parets d'aquesta caixa (l'algoritme del punt 6.4.2.1 explica els detalls d'aquest posicionament).

Finalment, es renderitza la geometria continguda dins el node des de cadascun dels punts de vista seleccionats. Les imatges generades es guarden directament com a textures gràcies a l'ús de l'extensió *OpenGL Frame Buffer Objects (FBOs)* [28, 29], que permet fer *render-to-texure*. De fet, s'utilitza la classe ja existent *FrameBufferObject* que permet gestionar còmodament l'ús d'aquesta extensió. Pel que fa a la dimensió d'aquestes textures, també es discuteix al punt 6.4.2.1.

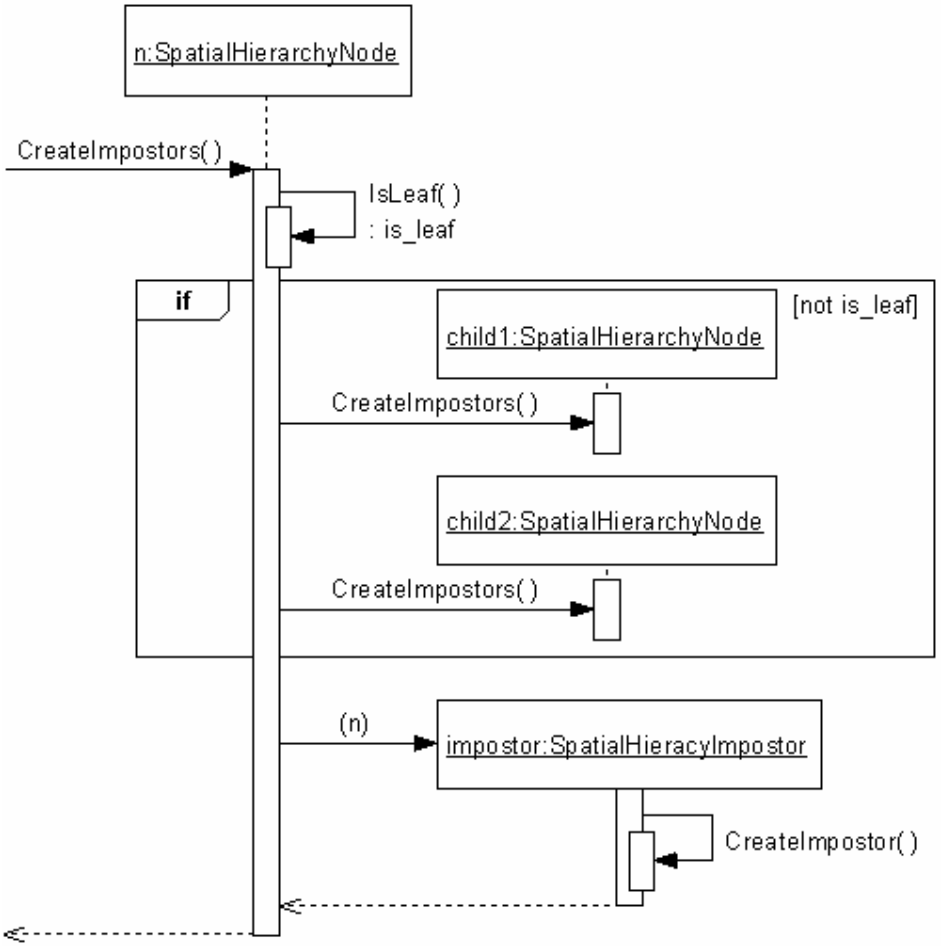


Fig. 6.4.1.2: *Diagrama de seqüència del procés de generació dels impostors sobre els nodes de la jerarquia.*

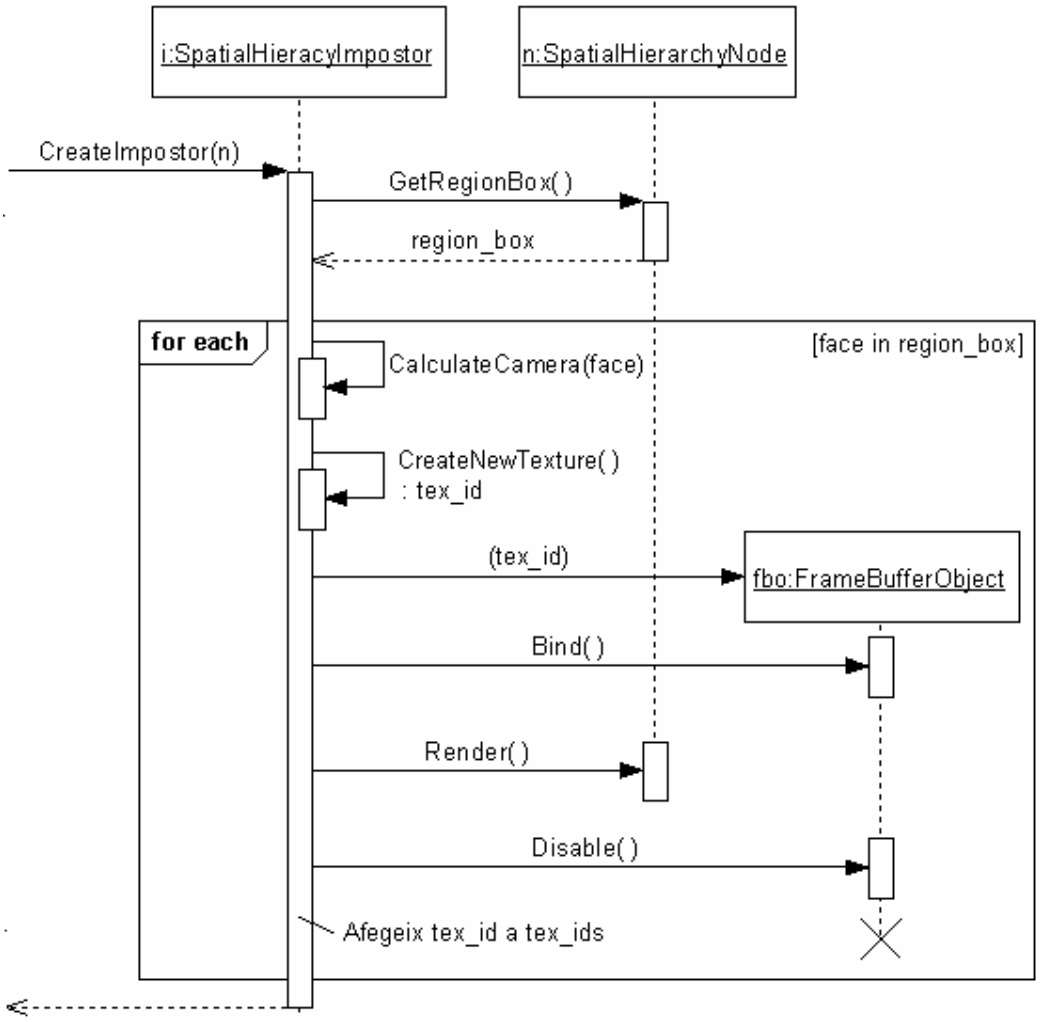


Fig. 6.4.1.3: *Diagrama de seqüència de la construcció d'un impostor.*

Una vegada s'han construït els impostors, aquests s'incorporen en el procés de visualització per tal que siguin utilitzats quan la geometria que representen signifiqui una petita porció de la imatge final. En particular, durant el recorregut pel *K-d tree* que fa l'algoritme *coherent hierarchical culling*, es comprova si el nombre de píxels que deixa la petjada d'un node de l'arbre és inferior a un cert llindar. En cas afirmatiu, es renderitzen directament els impostors del node i es finalitza el tractament d'aquest (no es llancen *occlusion queries* ni es visiten els seus fills). L'algoritme explicat al punt 6.4.2.2 explica com es calcula aquesta petjada i en el disseny d'experiments i l'anàlisi dels resultats obtinguts (punts 7 i 8), es discuteix el valor del llindar utilitzat.

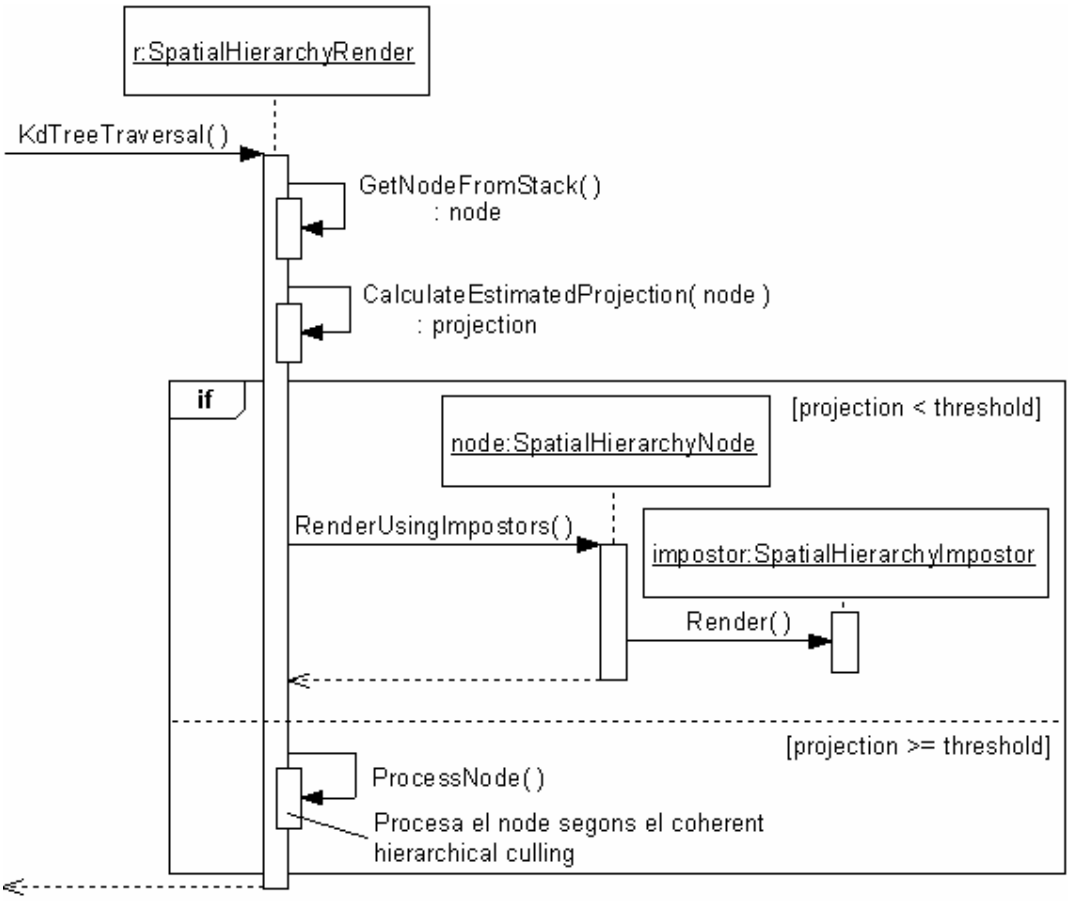


Fig. 6.4.1.4: *Diagrama de seqüència del procés de rendering amb impostors.*

6.4.2 Principals algoritmes

6.4.2.1 Generació dels impostors

Un dels aspectes més destacables relatiu a la generació d'impostors és la definició de la camera utilitzada per obtenir les imatges. D'entrada, es situa el *view reference point* (*VRP*) al centre de cadascuna de les cares de la caixa englobant d'un node de l'estructura jeràrquica. També es pren com a direcció de visió la normal a cada cara en sentit interior, és a dir, de manera que apunti cap al centre de la caixa.

Respecte al tipus de projecció, se n'utilitza una de tipus axonomètric. En aquest cas, el *frustum* de visió es pot ajustar molt millor a la caixa contenidora d'un node del *K-d tree* que no una projecció *perspectiva* (veure figura 6.4.2.1.1). Això permet que tota la resolució de la imatge generada es destini a la visualització del node, i que per tant, no es desaprofitin les regions de la perifèria de la imatge

(veure figura 6.4.2.1.2). A més, la projecció axonomètrica no requereix trobar a quina distància es troba l'observador del node (és indiferent perquè els raigs de visió són paral·lels i la imatge resultant no canvia si aquest s'acosta o s'allunya).

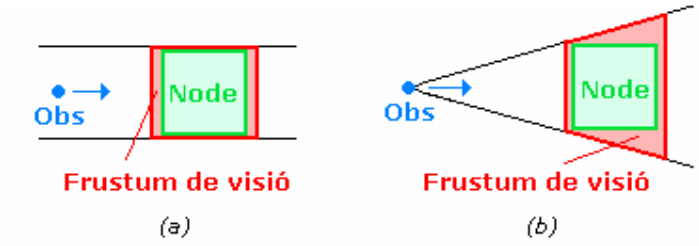


Fig. 6.4.2.1.1: *Adaptació del frustum de visió a la caixa contenidora d'un node del K-d tree.*
(a) Projecció axonomètrica (b) Projecció perspectiva

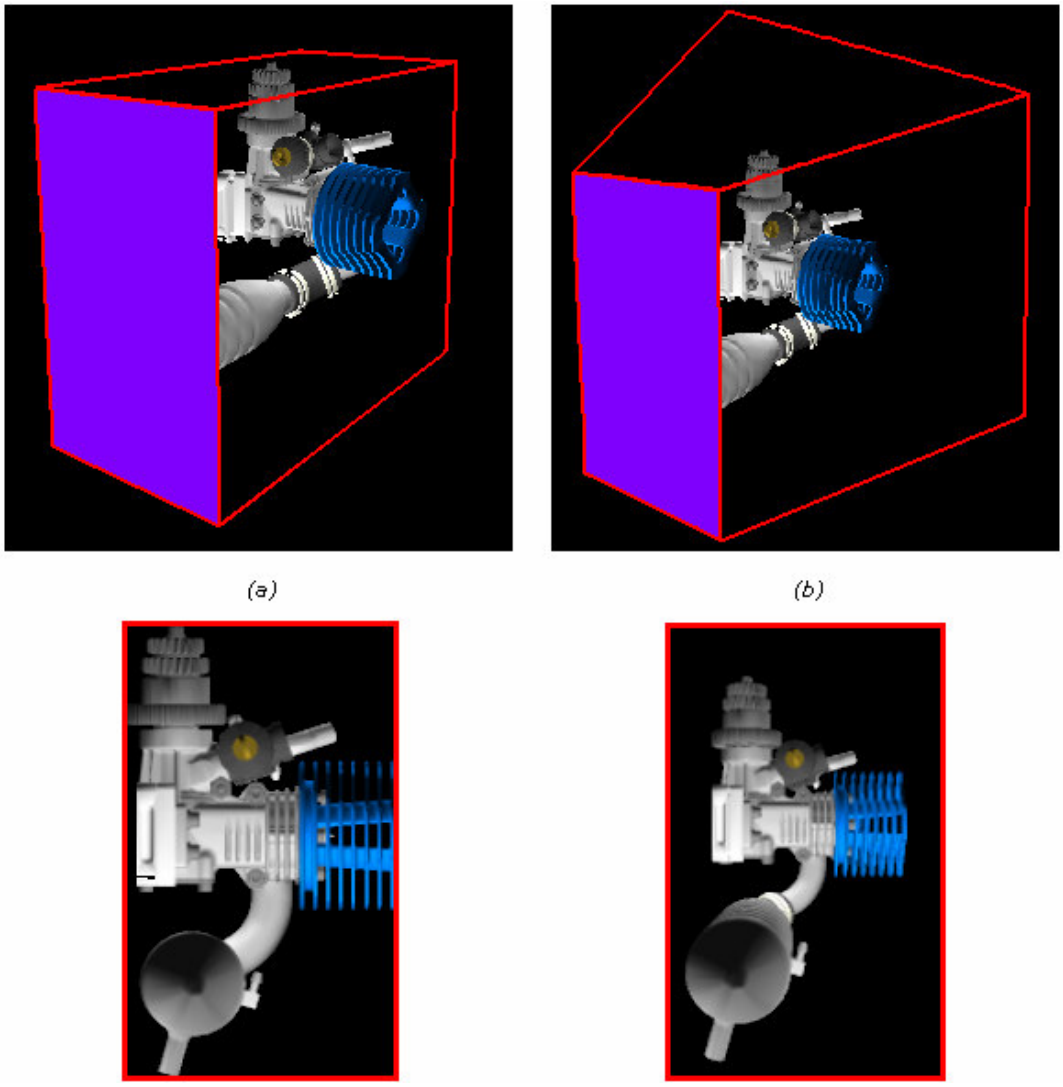


Fig. 6.4.2.1.2: *Impostor renderitzat amb diferents tipus de projecció.*
(a) Projecció axonomètrica (b) Projecció perspectiva

Abans de completar la definició de la camera, encara resta assignar el valor adequat als plans de retallat. Això es fa de manera que s'ajustin completament a la regió corresponent al node de l'estructura jeràrquica del qual s'està generant l'impostor. Així el frustum de visió pràcticament es correspon amb la caixa contenidora de la regió de l'espai que s'està representant.

Per altra banda, també cal determinar quina és la resolució de les textures utilitzades per a renderitzar els polígons impostors. Aquesta depèn del valor llindar que determina quan es poden utilitzar i de la relació d'aspecte de la cara corresponent de la caixa contenidora. Addicionalment, es té la restricció que les dimensions de les textures han de tenir valors múltiples de 2 (imposada per les característiques del hardware gràfic de *NVIDIA*).

D'aquesta manera, es pot prendre el supòsit que l'àrea màxima d'una textura impostor (cara de la caixa contenidora) és igual al valor del llindar corresponent. Per tant, una vegada fixat aquest llindar, es pot calcular quin tamany (en nombre de pixels) hauria de tenir una textura impostor amb relació d'aspecte 1. Tot i això, es prenen unes dimensions una mica més grans i així s'espera poder evitar problemes relacionats amb la texturació de polígons deformats per la projecció.

Finalment, donat que les cares de les regions del *K-d tree* no han de ser forçosament quadrades, es modifica la relació d'aspecte de la textura usant la de la cara corresponent. En aquest punt, és on entra en joc la restricció de dimensions múltiples de 2. Per a poder satisfer-la, aquesta modificació de tamany es fa de forma discreta, a partir de definir intervals tancats de valors de l'*aspect ratio*.

6.4.2.2 Càlcul del tamany de la projecció d'un node a la pantalla

La mida de la projecció (mesurada en pixels) sobre el pla de la pantalla que deixa la caixa contenidora d'un node de l'estructura jeràrquica es calcula de forma aproximada. Es segueixen els següents passos:

1. Es calcula l'àrea de tres cares de la caixa englobant, perpendiculars entre si. Només es consideren tres de les 6 cares perquè per cada parell de cares paral·leles, una sempre queda completament tapada per la resta.

2. S'obté la projecció de les tres àrees sobre el pla que passa pel punt mig de la caixa englobant i que té per normal la direcció de visió de la camera (pla de projecció). Això es computa mitjançant la següent expressió (on α és l'angle entre el vector normal a la cara i la direcció de visió):

$$A_{projectada} = A_{cara} * cos(\alpha)$$

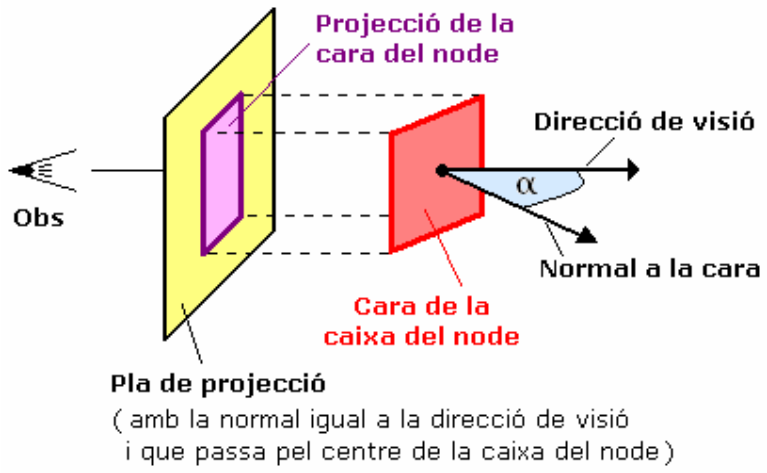


Fig. 6.4.2.2.1: *Càlcul de les àrees projectades sobre el pla de projecció.*

Com que els plans de les cares de la caixa contenidora sempre tenen l'orientació d'una de les tres direccions coordenades, es calculen els tres cosinus de les seves normals amb la direcció de visió. Així s'evita tornar-los a computar per cadascun dels nodes de l'arbre.

També cal considerar que amb aquests càlculs s'està fent una projecció paral·lela. Donat que la camera treballa amb una vista perspectiva, s'està cometent un cert error (per això es parla d'un valor aproximat). Tot i això, aquest error serà mínim, ja que al projectar sobre el punt mig de la caixa, la distància d'aquesta projecció és molt curta.

3. Es tornen a projectar les tres àrees, des del pla que passa pel centre de la caixa, fins al pla *z-near* (que es correspon amb el pla de la pantalla). En aquesta ocasió però, s'utilitza una projecció paral·lela, de manera que no es comet error en la conversió.

4. Es transforma el valor de la suma de les tres areas. Primer es converteix a coordenades normalitzades utilitzant les dimensions del *frustum* de visió, i després és passa a pixels mitjançant la resolució del *viewport* definit.

7 DISSENY D'EXPERIMENTS

Una vegada s'ha completat la fase d'implementació de les optimitzacions escollides, cal quantificar quin guany s'ha obtingut en el rendiment de l'aplicació. També s'ha de verificar si realment hi ha una correspondència amb el que s'havia pronosticat a l'anàlisi d'alternatives. Per tal d'assolir aquests propòsits, es preveuen realitzar una sèrie d'execucions sobre l'aplicació. Aquestes permetran obtenir dades numèriques, amb les quals es podrà valorar objectivament quin és l'impacte produït pels canvis realitzats. En els següents apartats, es descriuen les condicions experimentals amb les que es realitzen aquestes proves, s'enumeren les variables d'interès considerades i es proporciona una llista amb tots els experiments a realitzar.

7.1 Condicions experimentals

Les execucions que es preveuen fer tenen com a principal objectiu mesurar les diferències de rendiment entre l'aplicació original i la versió modificada amb les millores implementades. Addicionalment, també es vol determinar quin és l'impacte individual de cada optimització i es desitja contrastar el guany de les diferents configuracions d'alguns dels paràmetres d'aquestes.

Així doncs, com que la finalitat d'aquestes proves és fer comparacions, és imprescindible que aquestes es desenvolupin sota les mateixes condicions. Per tant, per poder creuar els valors de dues execucions, cal replicar amb exactitud totes les variables externes que poden influir en el resultat de la prova. En aquesta ocasió, s'han considerat les següents variables externes: el tipus de navegació, els models carregats i el hardware utilitzat.

7.1.1 Tipus de navegació

El sistema de navegació d'*Alice* generalment es controla fent ús del ratolí. Això, dona molta flexibilitat a l'usuari quan s'ha de moure per un entorn virtual, però fa que sigui gairebé impossible repetir amb exactitud la mateixa prova. Cal tenir en compte que el ratolí és un dispositiu que genera una entrada que és pràcticament contínua sobre el pla (realment té una precisió discreta, però aquesta és molt petita, de l'ordre d'un pixel).

D'aquesta manera, esdevé necessari disposar d'un mecanisme que permeti parametritzar el recorregut de la navegació, evitant així haver de fer ús del ratolí. Afortunadament, *Alice* ja té implementat un sistema de *plugins* que permet utilitzar múltiples modes de navegació. De fet, en les primeres versions de l'aplicació, ja s'havia desenvolupat un mode que desplaçava la camera pel model, a partir d'anar seguint trajectòries predefinides. Malauradament, en el pas a noves revisions, aquesta funcionalitat es va perdre i actualment no es troba disponible.

Donat que un mode de navegació d'aquestes característiques és idoni per a les proves que es volen fer, se n'ha creat un de nou. Aquest, parteix d'una entrada consistent en una llista de punts i va movent la posició de l'observador des del primer fins a l'últim, passant en ordre pels intermedis. Pel que fa a la direcció de visió, aquesta pren com a referència el següent punt de pas. A més, quan s'assoleix un d'aquests punts, per tal de fer suau la transició cap al següent, s'interpola durant uns quants frames el canvi de direcció de visió. Això evita que es produeixin salts no desitjables en l'animació resultant.

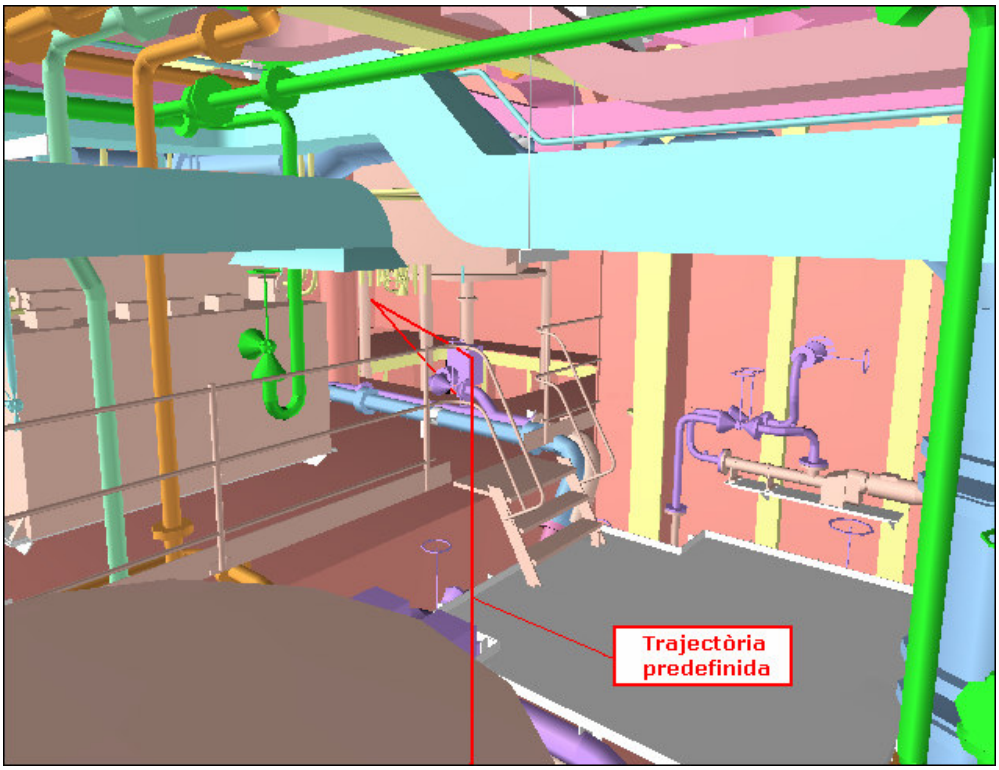


Fig. 7.1.1.1: *Mode de navegació basat en una trajectòria predefinida.*

Així doncs, amb aquest nou mode, s'aconsegueix reproduir un recorregut idèntic en execucions diferents. Això fa que si es volen comparar dues proves, a nivell de navegació només farà falta que tinguin la mateixa trajectòria d'entrada. Addicionalment, aquest mode de navegació manté constant el nombre de frames.

7.1.2 Models carregats

Com és evident, cada conjunt de proves que es vulgui sotmetre a comparació haurà de realitzar-se utilitzant el mateix model poligonal. A més a més, per a verificar si els guanys obtinguts depenen de les característiques intrínseques d'aquests, es pren un conjunt de models amb propietats ben diverses: moltes poques oclusions, amb / sense textures, gran quantitat d'objectes simples / pocs objectes detallats, ...

Per altra banda, també resulta d'especial interès mesurar el rendiment de l'aplicació quan es visualitzen models d'elevada complexitat. En aquests casos, s'espera que el guany obtingut amb les optimitzacions s'apreciï de forma clara, ja que la major part del temps de generació d'un *frame* s'invertirà processant geometria i enviant-la a pintar (és on incideixen les millores realitzades).

Tot i això, els models disponibles tenen una quantitat de polígons relativament petita (el més gran té unes 82.000 primitives) i insuficient per aquest tipus de proves. Per a resoldre aquesta mancança, es decideix replicar diverses vegades la geometria d'un dels models, situant les diferents copies una al costat de l'altra. Aquesta estratègia permet multiplicar el tamany dels models, fins al punt d'obtenir-ne de les dimensió desitjada.

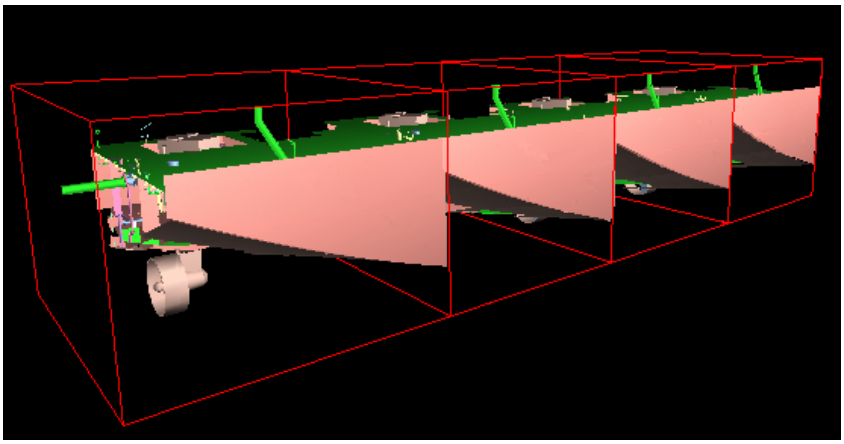


Fig. 7.1.2.1: *Model del bloc d'un vaixell replicat 4 cops i situant les copies, una al costat de l'altra.*

7.1.3 Hardware utilitzat

Totes les proves fetes en el marc d'aquest treball s'han realitzat utilitzant el mateix equip. Es tracta d'un *PC* portàtil, marca *ASUS*, amb dos processadors que van a una velocitat de 1,66 GHz cadascun i disposa de 2 GB de memòria RAM. Té una targeta gràfica *NVIDIA GeForce Go 7300* amb 512 Mb de memòria (128 Mb reals i la resta obtinguts mitjançant memòria virtual) i suporta les especificacions del *Shader Model 3.0.*

7.2 Variables d'interès

Un altre aspecte que cal definir per a completar el disseny d'experiments és la informació que es guardarà per cada prova. Aquesta, serà utilitzada durant l'anàlisi de resultats i permetrà extreure conclusions sobre diferents aspectes del rendiment de l'aplicació. Així doncs, per cadascuna de les execucions que es realitzin, s'enregistraran les següents variables d'interès:

a. *Temps de cada frame:* Es guarda el temps necessari per a generar cadascun dels fotogrames de la navegació definida. Com que dues execucions de la mateixa navegació tindran exactament el mateix nombre de *frames*, es pot fer una comparació un a un dels temps enregistrats.

b. *Temps de càrrega del model:* Es mesura el temps que es requereix per a efectuar la lectura de cadascun dels models de prova. S'espera que aquest temps es vegi afectat negativament per culpa dels nous precàlculs que introdueixen les optimitzacions implementades. Per aquest motiu, es desglossa el temps total de càrrega, a partir de mesurar la latencia de cadascuna de les fases de les que consta. Així es pot determinar quins són els preprocessos més crítics.

c. *Dimensions de l'atlas de textura:* Pel cas de models texturats, s'enregistren les dimensions de l'atlas de textura construït i es relacionen amb la suma de tamanys de totes les textures del model. Això permet valorar de forma numèrica quina és la quantitat d'espai desaprofitat al construir l'atlas. Per tant, permet justificar si l'algoritme que distribueix les textures funciona correctament.

d. *Increment de la complexitat del model*: Tal i com s'ha explicat en el punt 6.3.1.1, la geometria original del model es sotmet a un procés de conversió que té com a resultat una malla de triangles (es triangulen les primitives existents). Addicionalment, per a gestionar correctament els modes d'adreçament de la texturació, es subdivideixen els triangles, de manera que totes les coordenades de textura es trobin compreses en el rang [0,1]. Així doncs, per poder valorar l'increment de complexitat introduït per aquests procediments, es comptabilitza el nombre de primitives del model original i la quantitat de triangles de la malla resultant.

En algunes situacions, també seria d'especial interès mesurar la qualitat de les animacions obtingudes. Cal pensar que algunes de les millores desenvolupades incrementen el rendiment de l'aplicació a partir de simplificar el *rendering* de la geometria de menys rellevància (més llunyana, molt tapada, ...). Tot i això, els indicadors existents per a mesurar objectivament aquesta qualitat són molt complexes i queden fora de l'abast d'aquest treball.

D'aquesta manera, s'opta per fer una valoració subjectiva de les animacions generades. Es considerarà que tenen la qualitat suficient quan no s'hi puguin apreciar *artifacts* a simple vista. Així doncs, mitjançant observacions d'aquestes característiques s'assignarà un valor als llindars que determinen quan un node està tapat (*occlusion queries*) o quan es poden començar a utilitzar impostors.

7.3 Experiments a realitzar

Les proves que s'han planificat s'estructuren segons el tipus de propòsit que persegueixen. D'aquesta manera, es preveu fer un grup de tests per cadascuna de les optimitzacions implementades. Aquestes proves tenen per objectiu determinar la millor configuració d'alguns dels paràmetres de cada millora. Finalment, es planteja una bateria de proves per comparar el guany obtingut en cadascun dels desenvolupaments, prenent com a referència el rendiment de l'aplicació original.

7.3.1 Estructura jeràrquica de divisió de l'espai

Els experiments relacionats amb la implementació d'un *K-d tree* es centren en la qualitat de les divisions realitzades. S'executarà l'algoritme de construcció de la jerarquia sobre diferents models i es comprovarà si els plans que parteixen les

regions de l'espai es seleccionen correctament (minimitzant el nombre d'objectes tallats i a la vegada distribuint equitativament la geometria). Aquesta valoració es farà de forma gràfica a partir de pintar amb filferros, la caixa contenidora de cadascuna de les regions en què s'ha dividit l'espai de l'escena.

S'usarà un criteri de fulla consistent en un màxim de 20.000 vèrtexs per regió i 20 nivells de profunditat de l'arbre. La tolerància que permet distribuir la geometria de forma que no sigui completament equilibrada és d'un 10%.

S'ha agafat un conjunt de models amb característiques molt diferenciades. Així, es podrà verificar que l'algoritme funciona correctament en tots els casos. En particular, les escenes utilitzades són: *Motorcillo.P3D*, *Astano.P3D*, *Texaco.P3D*, *NoPassaran.P3D* i *CatedralCadiz.P3D*.

7.3.2 Hardware occlusion queries

L'únic paràmetre pendent de configuració i que està directament relacionat amb aquest desenvolupament és el llindar de visibilitat. Indica a partir de quin nombre de pixels que superen el test *z-buffer* (resultat d'una *hardware occlusion query*), la geometria d'un node es considera visible i per tant, s'ha de renderitzar. Si aquest valor és molt gran, es simplificarà molta geometria, però també poden aparèixer *artifacts* de *cracking* relacionats amb els objectes tapats. Per contra, si el valor és molt petit, les imatges seran de més qualitat, però no es detectaran gaires oclusions i per tant, el guany obtingut serà molt menor.

D'aquesta manera, el propòsit d'aquestes proves és calcular un valor òptim per a aquest llindar. Es realitzaran 4 execucions d'una navegació de 1.000 frames sobre el model *Texaco.P3D* (bloc d'un vaixell), replicat 3 cops i disposant els blocs de forma consecutiva. Amb una escena d'aquestes característiques, s'espera que hi hagi un grau d'oclusió notable degut a la voluminositat dels elements que la composen (parets entre compartiments, tubs, equips, ...). A cadascuna de les execucions, s'utilitzarà un valor diferent per al llindar de visibilitat. Els valors que es prendran són 1, 100, 200 i 300.

7.3.3 Vertex Buffer Objects

El renderitzat de la geometria mitjançant *VBOs* és una millora que depèn d'un gran nombre de paràmetres. En aquesta ocasió, es pretén utilitzar una sèrie de tests per a calcular el valor més adequat per a cadascun d'ells. Així doncs, les variables d'entrada que es volen estudiar són les següents:

1. *Tamany màxim*: Es tracta del nombre màxim de vèrtexs continguts en un mateix *VBO*. Com que els *VBOs* es generen basant-se en l'estructura jeràrquica, aquest valor ve determinat pel criteri de fulla del procés que la construeix. Els millors rendiments s'obtenen utilitzant *VBOs* molt grans, sempre hi quan no es superi un cert límit que depèn de les característiques de la targeta gràfica. S'estima que aquest valor límit es troba al voltant d'uns 50.000 vèrtexs. D'aquesta manera, es faran execucions utilitzant tamanys que siguin d'aquest ordre de magnitud. Les dimensions que s'han seleccionat són: 20.000, 40.000, 60.000, 80.000 i 100.000.

2. *Tipus de dades*: S'especula que el rendiment dels *VBOs* pugui veure's influenciat pel tipus de dades amb què es representen els components de la informació dels vèrtexs. En particular, es tracta del format que s'usa per a guardar el color. Es vol constatar si utilitzant *unsigned bytes* s'obtenen millors resultats que fent servir *floats*. Per aquest motiu, es plantejen dos tests: un emmagatzemant el color amb *unsigned ints* i l'altre enregistrant-lo mitjançant *floats*.

3. *Organització de les dades*: Aquestes proves fan referència a la distribució de la informació dels vèrtexs en els *buffers* de la targeta gràfica. D'entrada, es planteja fer ús d'un *buffer individual* per a cada component (posició, color, normal i coordenades de textura), o bé generar un únic *buffer* per a totes les dades. A més, en el cas que es prengui aquesta segona alternativa, es pot escollir com distribuir la informació de les components dins el *buffer*: separant-les o disposant-les de forma intercalada. Es té la sospita que un únic *buffer* amb les dades intercalades és la configuració que dona millors resultats, però s'efectuarà una prova amb cada organització per a poder verificar-ho.

4. *Ús de coordenades de textura*: Es vol comprovar com varia el rendiment de l'aplicació si no s'envien a la *GPU* les coordenades de textura, en els models

que no les utilitzen. És evident que la memòria de la targeta gràfica utilitzada pels diferents *buffers* serà menor, però també es vol estudiar si això accelera el procés de *rendering*. Per tant, es planifica fer dos experiments de característiques pràcticament idèntiques, on l'única diferència es trobi en l'ús de les coordenades de textura.

L'escena que es farà servir per a executar totes aquestes proves és la del model *Texaco.P3D*, però afegint-li 10 copies del model original (una al costat de l'altre). D'aquesta manera, s'espera poder obtenir un model de gran tamany, on la complexitat de la geometria en sigui el coll d'ampolla. En aquesta situació és on els *VBOs* produeixen un impacte més gran i com a conseqüència, les diferencies entre configuracions es podran apreciar amb molta més facilitat.

Pel que fa a la navegació, s'ha escollit un recorregut que va de punta a punta del model, travessant tots els blocs del vaixell i que té una durada de 3.000 *frames*. Inicialment, pràcticament tota la geometria de l'escena es troba dins el *frustum* de visió i poc a poc, aquesta es va quedant darrera l'observador. Així doncs, s'espera que el temps necessari per a generar un *frame* es vagi reduint de forma progressiva

Donat que a cada grup de tests només s'estudia el comportament d'una única variable, cal fixar un valor constant per a la resta de paràmetres. Per defecte, s'agafa la següent configuració: tamany màxim de 50.000 vèrtexs, totes les components dels vèrtexs es representen com a *floats*, s'utilitza un únic *buffer* amb dades intercalades i les coordenades de textura s'envien a la *GPU*.

Finalment, no es pot obviar el tractament que s'aplica a les dades geomètriques per a poder adaptar-les a les necessitats dels *VBOs*. S'aprofitaran les execucions realitzades per a comprovar si les primitives es triangulen correctament. Addicionalment, es faran proves amb models que utilitzin textures (CatedralCadiz.*P3D i SantaMaria.P3D*) per a assegurar que l'atlas s'ha generat correctament i que la subdivisió segons les coordenades de textura no dona problemes.

7.3.4 Impostors

De forma similar a l'optimització de les *hardware occlusion queries*, les proves relatives a l'ús d'impostors tenen la finalitat d'ajustar el llindar corresponent, fins a

un valor que sigui òptim. En aquest cas però, el llindar determina quan un node pot visualitzar-se mitjançant impostors i quan no. Com a unitat de referència, es pren una estimació del nombre de pixels de la projecció d'un node de l'estructura jeràrquica en pantalla. Així doncs, aquí també cal balancejar el guany obtingut (gràcies a la simplicitat dels impostors), en relació amb la pèrdua de qualitat de les imatges produïdes (no es suporten efectes de *parallax, artifacts de cracking, ...*).

Cal tenir en compte que el valor del llindar que es busca, hauria de ser superior al del llindar de visibilitat. La representació mitjançant impostors, tot i tractar-se d'una simplificació, s'acosta més a la imatge real del que ho fa l'*occlusion culling* (senzillament suprimeix geometria). Per tant, és lògic esperar que els impostors es puguin utilitzar en un major nombre situacions. D'aquesta manera, els tests plantejats prenen valors per al llindar d'impostors lleugerament superiors als del llindar de visibilitat. Són els següents: 250, 500, 1000 i 2000.

El model utilitzat per aquests experiments, a diferència dels desenvolupaments previs, és el *Motorcillo.P3D*. Es fan 5 rèpliques de la geometria, però aquest cop es disposen de forma solapada (no s'aplica cap translació a les copies). Això permet incrementar la densitat de la geometria de l'escena, fent que el guany resultant al utilitzar impostors sigui molt més gran. Així s'espera poder veure amb més claredat quin és l'efecte que produeixen.

Respecte a la navegació utilitzada, en aquesta ocasió s'ha escollit un recorregut de només 150 *frames*. Es parteix d'un punt allunyat i es fa una aproximació al centre del model. Aquest esquema es repeteix diverses vegades, però a cadascuna es fa l'acostament des d'una direcció diferent. Es preveu que en els punts de vista més distants, totes les execucions utilitzin els impostors, mentre que en ubicacions més pròximes, només ho facin les de llindar més gran.

7.3.5 Tests generals

L'últim paquet de proves té com a principal objectiu comparar el guany obtingut per cadascuna de les millores implementades. Per assolir aquesta fita, s'efectuaran una sèrie de experiments que permetran contrastar l'increment de rendiment aportat per cadascuna. Es realitzaran execucions sobre l'aplicació original, sobre versions on s'activin individualment cadascun dels nous desenvolupaments i sobre un entorn on totes les millores es trobin operatives de forma simultània.

Per evitar que els resultats obtinguts depenguin del model carregat o de la navegació que es segueix, es repetiran els mateixos experiments sobre un conjunt d'escenes i trajectòries amb característiques ben diverses. Concretament, es preveu utilitzar: *Texaco.P3D* (original), *Texaco.P3D* (replicat 5 vegades), *Motorcillo.P3D*, *CatedralCadiz.P3D* i *SantaMaria.P3D*.

Addicionalment, es volen aprofitar les execucions realitzades sobre la versió final d'*Alice* (on totes les millores estan en funcionament), per a analitzar altres aspectes d'interès. Un d'ells és el temps de càrrega de l'aplicació. S'espera que aquest es vegi influenciat negativament ja que s'han afegit nous preprocessos. També es volen mesurar les dimensions del nou l'atlas de textura i es volen relacionar amb el sumatori de tamanys de totes les textures del model. Així es podrà veure quina quantitat d'espai es desaprofita. Finalment, es vol quantificar l'augment de complexitat de la geometria del model. La conversió a malla de triangles i la subdivisió segons les coordenades de textura fan que hi hagin moltes més primitives que en el model original.

8 RESULTATS OBTINGUTS

L'objectiu d'aquest capítol és mostrar els principals resultats obtinguts, una vegada s'han executat els tests plantejats en el punt anterior. Per cada grup de proves proposat, es proporcionen les dades més rellevants, ja sigui en format numèric o gràfic, i se'n fa un anàlisi valoratiu. D'aquesta manera, els següents quatre apartats es dediquen a les optimitzacions implementades, mentre que el cinquè fa un estudi comparatiu dels guanys obtinguts per cadascuna. Finalment, en el *CD* adjunt a aquesta memòria, s'hi poden trobar uns fulls de càlcul corresponents als resultats complets de totes les execucions realitzades.

8.1 Estructura jeràrquica de divisió de l'espai

Les proves relatives a la construcció de l'estructura jeràrquica consisteixen en verificar que la divisió de les regions de l'espai es fa correctament. El resultat d'aquestes execucions és satisfactori ja que els nodes generats s'adapten a la distribució de la geometria a través de l'escena. Així doncs, a la imatge de la figura 8.1.1, es pot veure amb facilitat que a les zones de més complexitat és on es subdivideix l'escena fins a màxima profunditat de l'arbre.

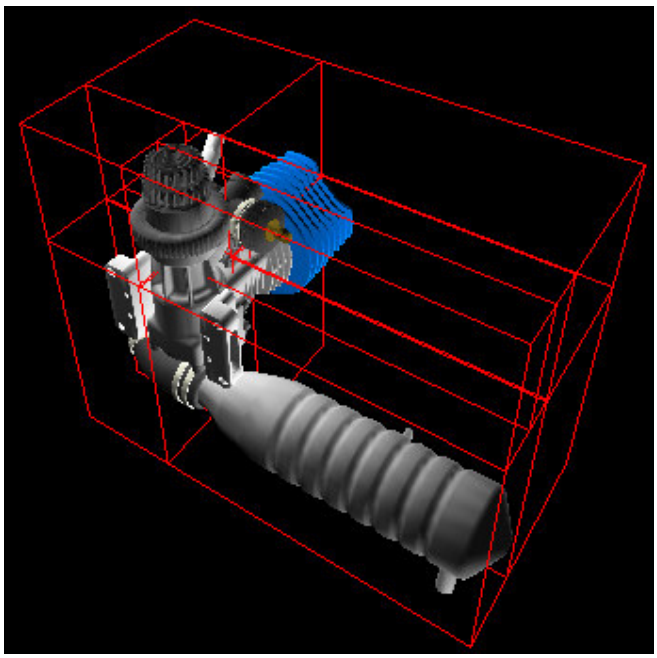


Fig. 8.1.1: *Divisió de l'espai en regions en el model Motorcillo.P3D.*

Tot i això, al tractar-se d'una escena on tota la geometria es troba disposada sense cap mena de separació, no es pot apreciar el moviment dels plans de tall per tal de minimitzar el nombre de triangles intersecats. Això no és així en el cas de la figura 8.1.2, on es mostra l'estructura jeràrquica d'un model consistent en objectes clarament espaiats.

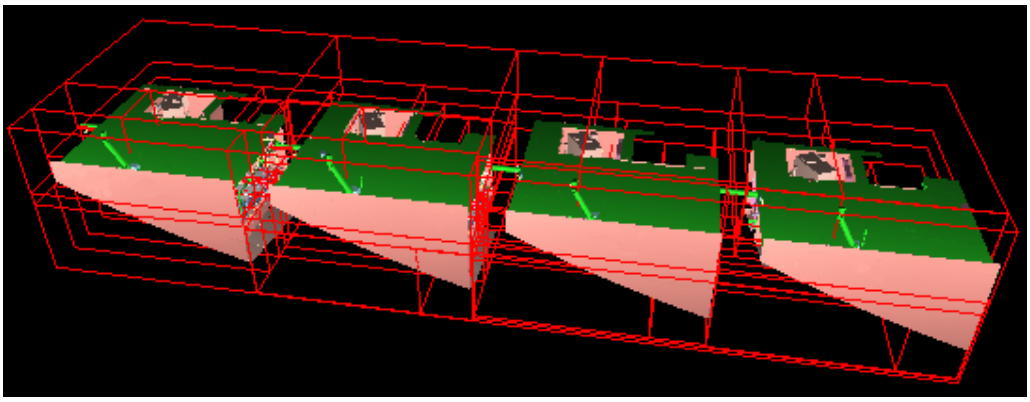


Fig. 8.1.2: *Divisió de l'espai en regions en el model Texaco.P3D. S'ha replicat 4 cops la geometria del model, de manera que cada copia queda perfectament espaiada.*

8.2 Hardware occlusion queries

Les execucions que es centren en l'optimització de les *hardware occlusion queries* tenen el propòsit de determinar el valor òptim per al llindar de visibilitat. Aquest llindar determina a partir de quin nombre de pixels que superen el *z-test* en una *occlusion query* ja no es pot aplicar *occlusion culling*. Per tant, cal trobar un valor que aconsegueixi un equilibri entre l'eficiència del procés de visualització i la qualitat de les imatges / animacions obtingudes.

Respecte al rendiment obtingut en cadascun dels tests, aquest es pot observar a la gràfica de la figura 8.2.1. S'hi aprecia amb claredat que com major és el valor del llindar de visibilitat, més ràpid es renderitza l'escena. Això era d'esperar ja que a mesura que el llindar és fa menys restrictiu (valors més grans), l'oclusion culling s'aplica en més situacions i per tant, es processa menys geometria.

Tot i això, també es detecta que el guany és cada vegada menor. Per adonar-se d'aquest fet, només cal fixar-se en les fases de la navegació on les gràfiques de diferents execucions es solapen. Per a llindars alts, aquest tipus d'etapes apareixen

amb més freqüència. La justificació d'aquesta situació es troba en que ja s'està eliminant tota la geometria que queda pràcticament tapada del tot i que per a eliminar-ne més, caldria ampliar el llindar fins a valors no raonables.

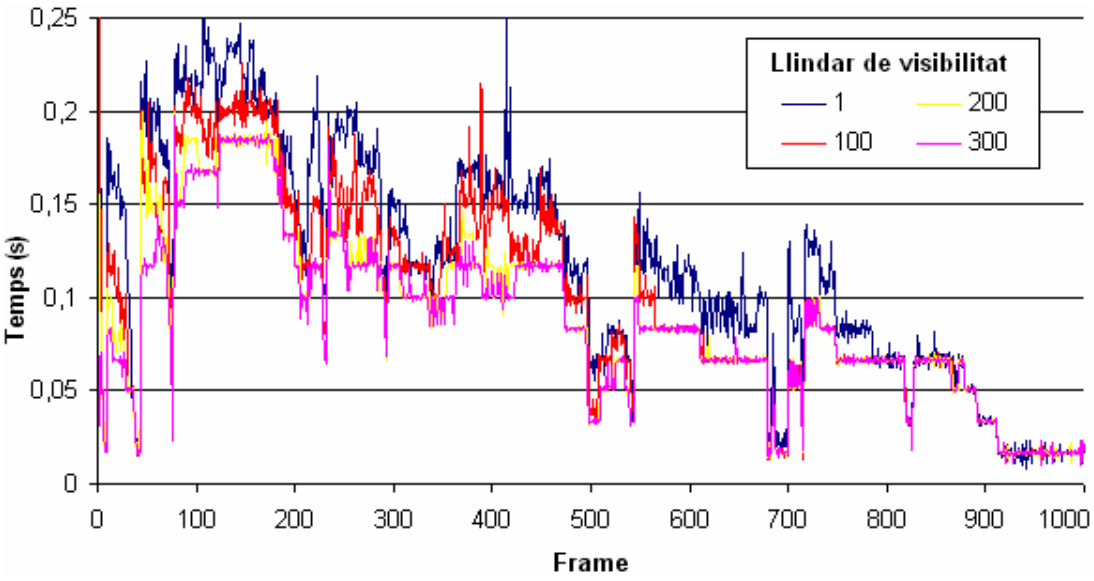


Fig. 8.2.1: *Gràfica que mostra el temps necessari per generar cadascun dels frames de la navegació, segons el valor del llindar de visibilitat.*

Per altra banda, també s'analitza la qualitat de les imatges obtingudes en cada cas. Quan el valor del llindar de visibilitat és 1, la *renderització* de l'escena és de màxima qualitat i no presenta cap mena d'artefacte. Si s'augmenta fins a 100, en alguns casos apareixen *artifacts* (pixels més foscos). De totes maneres, això passa en situacions relativament aïllades, que queden força desapercebudes (gràcies a l'animació), i que requereixen fixar-s'hi amb deteniment per a que siguin detectades. Per a valors de 200 i 300, aquests *artifacts* apareixen cada cop amb més freqüència i augmenten progressivament el seu tamany. Això ho fan fins al punt de que la visualització obtinguda no pugui considerar-se com a vàlida.

A la figura 8.2.2, es pot veure una comparativa de la imatge generada amb cada llindar en un mateix *frame* de la navegació. Es tracta d'un fotograma on hi ha un gran nombre de canonades que creuen el *frustum* de visió, tapant pràcticament del tot el que hi ha al fons. En el cas de valors de 1 i 100, la geometria del fons es renderitza i la qualitat resultant és màxima. Per contra, amb valors de 200 i 300, s'aplica *occlusion culling* fent que s'observi una regió de dimensió no despreciable de pixels de color negre (*artifact*).

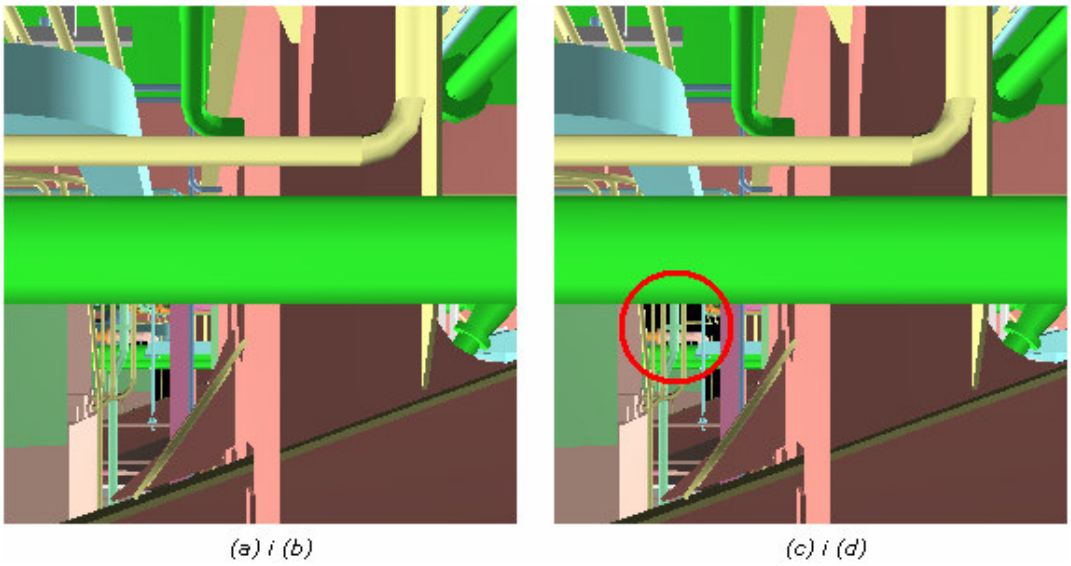


Fig. 8.2.2: *Artifacts que apareixen a les imatges generades, en funció del llindar de visibilitat.*
(a) Valor 1 (b) Valor 100 (c) Valor 200 (d) Valor 300

En base a aquests resultats, s'escull un valor de 100 per al llindar de visibilitat. Es considera que el guany obtingut (en relació amb un valor de 1) compensa la pèrdua mínima en la qualitat de la imatge. Pel que fa a les alternatives de major rendiment (200 i 300), aquestes es desestimen directament degut a la magnitud de l'error que introdueixen.

8.3 Vertex Buffer Objects

Les execucions relatives a la utilització de *VBOs* persegueixen el valor òptim de cadascun dels paràmetres de configuració d'aquesta millora. En els següents apartats, s'analitzen els resultats obtinguts amb el conjunt d'experiments destinats a estudiar de forma individual cada paràmetre. Finalment, en l'últim punt es fa una valoració dels processos que modifiquen la geometria del model i que s'han incorporats en la implementació dels *VBOs*.

8.3.1 Tamany màxim

La dimensió dels *VBOs* generats depèn directament de la subdivisió de l'espai realitzada a l'estructura jeràrquica. Per tant, el tamany màxim d'un *VBO* ve fixat pel criteri de fulla del *K-d tree*. Donat que aquests incrementen el seu rendiment a

mesura que s'augmenta el seu tamany (fins a un cert límit fixat per la targeta gràfica), interessa fer-los tan grans com es pugui, però intentant conservar a la vegada, un mínim d'adaptabilitat en l'estructura jeràrquica. D'aquesta manera, les execucions realitzades pretenen trobar amb quin valor es pot renderitzar de forma més eficient l'escena. A la figura 8.3.1.1, es mostra una gràfica amb l'evolució del temps de generació d'un *frame* en cadascun dels casos:

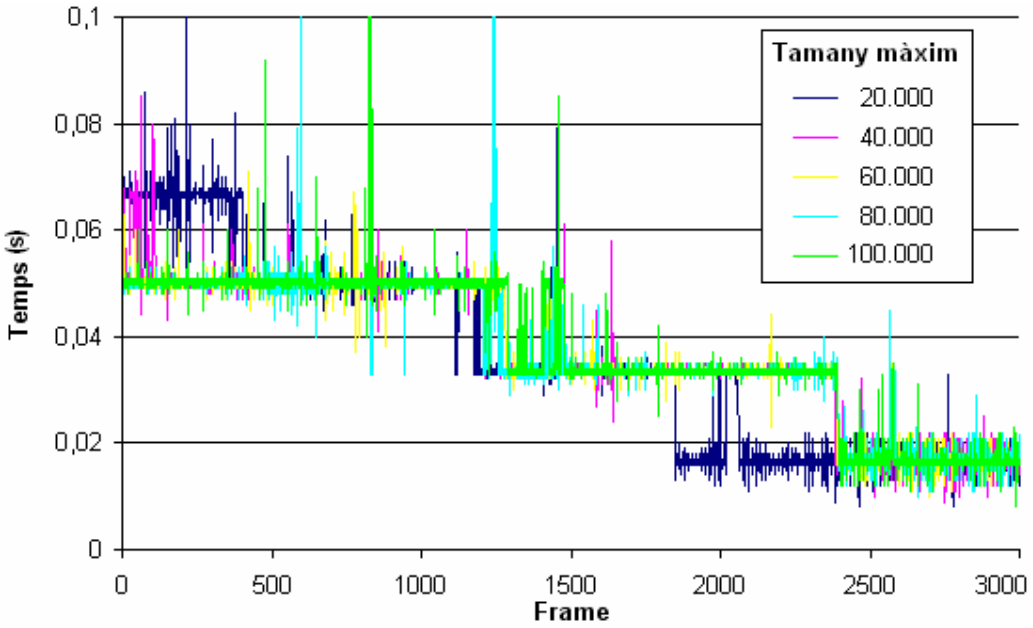


Fig. 8.3.1.1: *Gràfica comparativa del rendiment obtingut amb diferents tamanys dels VBOs.*

D'entrada, s'observa una reducció evident del temps de *frame*, a mesura que evoluciona la navegació. Això encaixa perfectament amb el fet que cada vegada hi ha menys geometria dins el *frustum* de visió, tal hi com s'havia pronosticat. Tot i això, les gràfiques obtingudes tenen una aparença d'escala: es mantenen gairebé sempre constants, excepte en uns punts molt concrets on es produeix un descens sobtat. Aquest fet està relacionat amb la reducció de geometria que es processa, però no és directament atribuïble als nodes de l'estructura jeràrquica (el *Kd-tree* té moltes més regions que no les gràfiques graons). Per tant, està clar que hi ha un factor, probablement relacionat amb algun paràmetre del hardware gràfic, que està determinant aquest tipus de comportament. Caldria fer més proves per a determinar-ho amb precisió. De totes maneres, com que *Alice* està pensat per a ser executat des de diferents plataformes, no resulta d'interès optimitzar-lo en funció d'un hardware específic i per aquest motiu, s'ha obviat aquesta qüestió.

Passant a comparar els rendiments obtinguts entre les diferents execucions, els *VBOs* de 20.000 vèrtexs són els que més es diferencien de la resta. Són una mica més ràpids quan es processa poca geometria, tot i que també són més lents quan se'n renderitza molta. Pel que fa a la resta de tamanys (40.000, 60.000, 80.000 i 100.000), aquests aconsegueixen uns resultats pràcticament idèntics. De fet, l'única desviació destacable l'ofereixen els de dimensió 40.000, els quals necessiten més temps per a generar els primers fotogrames de la navegació. En qualsevol cas, es té la impressió que repetint els tests amb models de major tamany es podrien extreure dades més concloents. Malauradament, l'elevat temps que fa falta per a carregar escenes d'aquestes característiques (s'han introduït preprocessos molt costosos a la lectura de models) fa que això no sigui viable.

D'aquesta manera, s'escull un valor de 50.000 com a nou tamany per als *VBO*. S'especula que així s'obté un comportament similar a les mides de *VBOs* de la majoria de proves, sense deixar de banda la capacitat d'adaptació de l'estructura jeràrquica. A més, l'elecció realitzada coincideix amb el valor teòric que s'havia extret de la bibliografia, cosa que reafirma encara més la decisió que s'ha pres.

8.3.2 Tipus de dades

Es creu que el tipus de dades utilitzat per a enregistrar la informació dels vèrtexs pot influir en l'eficiència del procés de visualització. Es tenen indicis, almenys en hardware més antic, que les targetes gràfiques processen més ràpidament els *unsigned bytes* que els *floats*. De totes maneres, també es creu que el hardware més modern ja no està subjecte a aquesta limitació.

Així doncs, es planteja emmagatzemar la component de color dels vèrtexs mitjançant *unsigned bytes*. Es vol comprovar si això afecta d'alguna manera al temps de generació de les imatges de l'escena. Per fer-ho, s'han realitzat dos tests, canviant el tipus de la informació de color entre *float* i *unsigned byte*.

La gràfica de la figura 8.3.2.1 recull els resultats obtinguts. No es detecten pràcticament diferències entre les dues execucions, a excepció de l'interval comprès entre els *frames* 800 i 1.200, on els *unsigned bytes* obtenen un rendiment superior. Tot i això, donat que aquesta diferència no segueix una regularitat, es té la sospita no la genera el tipus de dades, sinó que apareix per efecte d'algun altre factor desconegut.

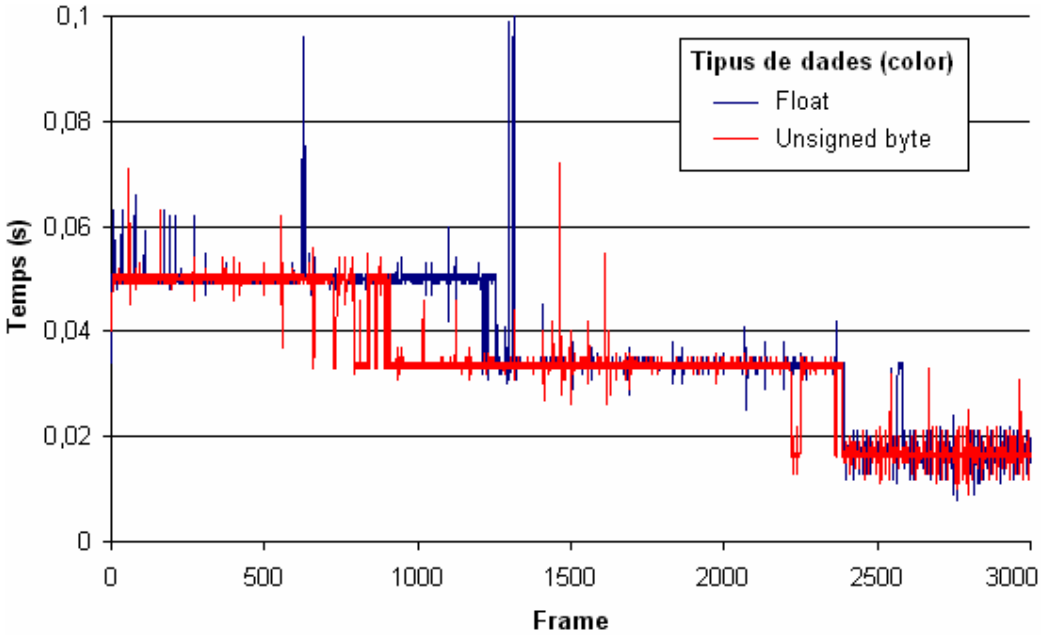


Fig. 8.3.2.1: *Gràfica comparativa del rendiment obtingut utilitzant diferents tipus de dades per a representar el de color dels vèrtexs.*

D'aquesta manera, no es considera que usant *unsigned bytes* s'obtingui un guany notable respecte als *floats*. Cal destacar que aquests últims ocupen 4 cops més memòria que els primers, però no es necessita alinear-los per a treballar amb processadors de 32 bits. Per tant, amb la intenció de mantenir la homogeneïtat amb la resta de components, es decideixen utilitzar els *floats*.

8.3.3 Organització de les dades

Els *VBOs* ofereixen la possibilitat d'organitzar la informació dels vèrtexs de diferents maneres. Es permeten posar totes les dades en un únic *buffer* o crear un *buffer* independent per cada component. A més, en el cas que s'usi un sol *buffer*, es dona l'opció d'agrupar la informació de cada component en blocs, o bé de disposar-la de forma intercalada.

La literatura referent a la matèria indica que fent servir un únic *buffer* i intercalant les dades de cada component és com s'obtindran millors resultats. Tot i això, s'ha fet una execució amb cadascuna de les tres configuracions per tal de comprovar-ho. La figura següent en mostra els resultats:

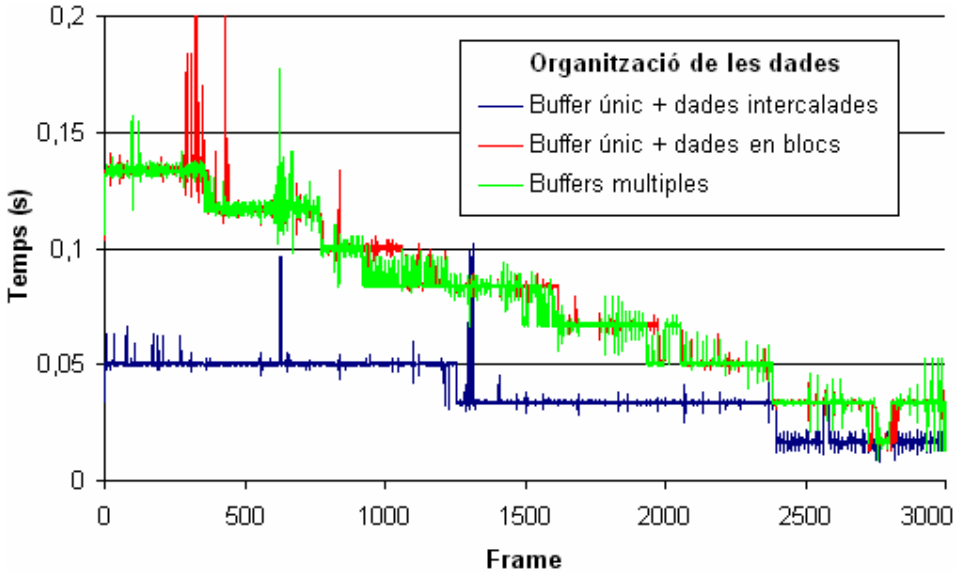


Fig. 8.3.3.1: *Gràfica comparativa del rendiment obtingut distribuint la informació dels vèrtexs en diferents configuracions de buffers.*

La gràfica anterior confirma de forma indiscutible la intuïció que es tenia a priori. La opció d'un únic *buffer* és clarament més ràpida que les altres dues alternatives, les quals tenen un rendiment similar. Per tant, aquesta serà la configuració que es prendrà com a definitiva.

Per altra banda, no es pot deixar de comentar la forma de les gràfiques de les dues distribucions que s'han descartat. Aquestes també tenen un aspecte d'escala, però amb molts més graons que les execucions on s'intercalen les dades. En aquest cas, si que sembla que el pas de cada graó estigui determinat pels nodes l'estructura jeràrquica (deixen de pintar-se al avançar la navegació). A més, l'alçada d'aquests graons és la mateixa que en la gràfica de l'opció triada. Això porta a pensar que en aquests casos no s'està aplicant algun tipus d'optimització. Aquesta s'encarregaria d'aglutinar el cost de rendering de diversos *VBOs* en el cost necessari per processar-ne un de sol.

8.3.4 Ús de les coordenades de textura

Existeix un gran nombre d'escenes que no utilitzen les funcionalitats de la texturació. Davant d'aquestes situacions, les coordenades de textura esdevenen inservibles i per aquest motiu, es planteja eliminar-les dels *VBO*. Això suposa fer una gestió addicional i per això, interessa determinar quin és el benefici obtingut i

poder valorar així si compensa. D'aquesta manera, es porta a terme un test que comparen quina diferència hi ha entre incloure les coordenades de textura al *VBO* i no fer-ho. La gràfica de la figura 8.3.4.1 en mostra els resultats:

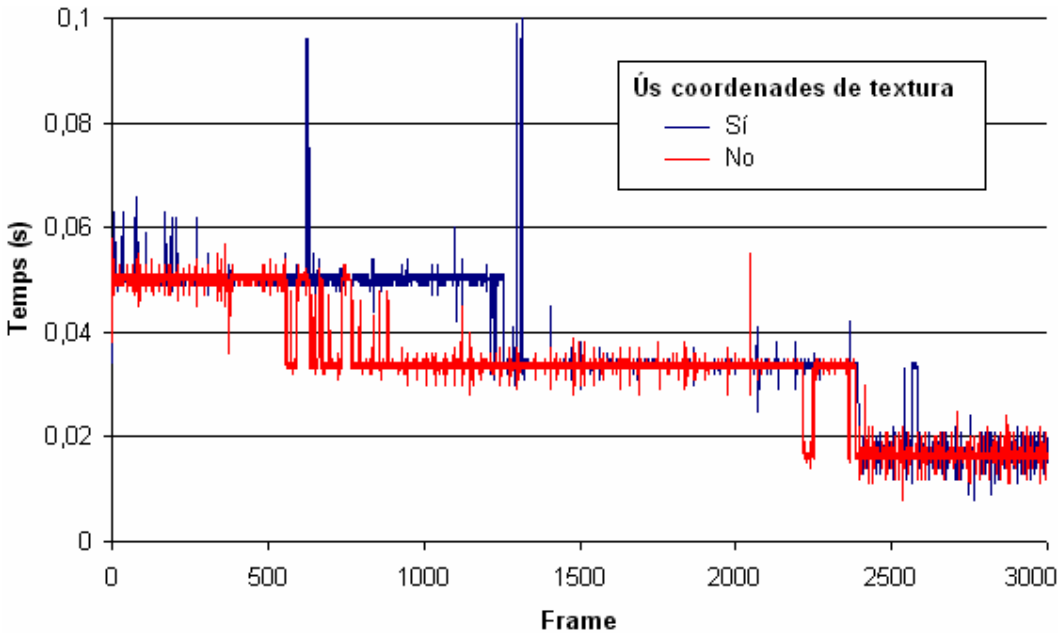


Fig. 8.3.4.1: *Gràfica comparativa del rendiment obtingut entre quan s'envien les coordenades de textura i quan no es fa.*

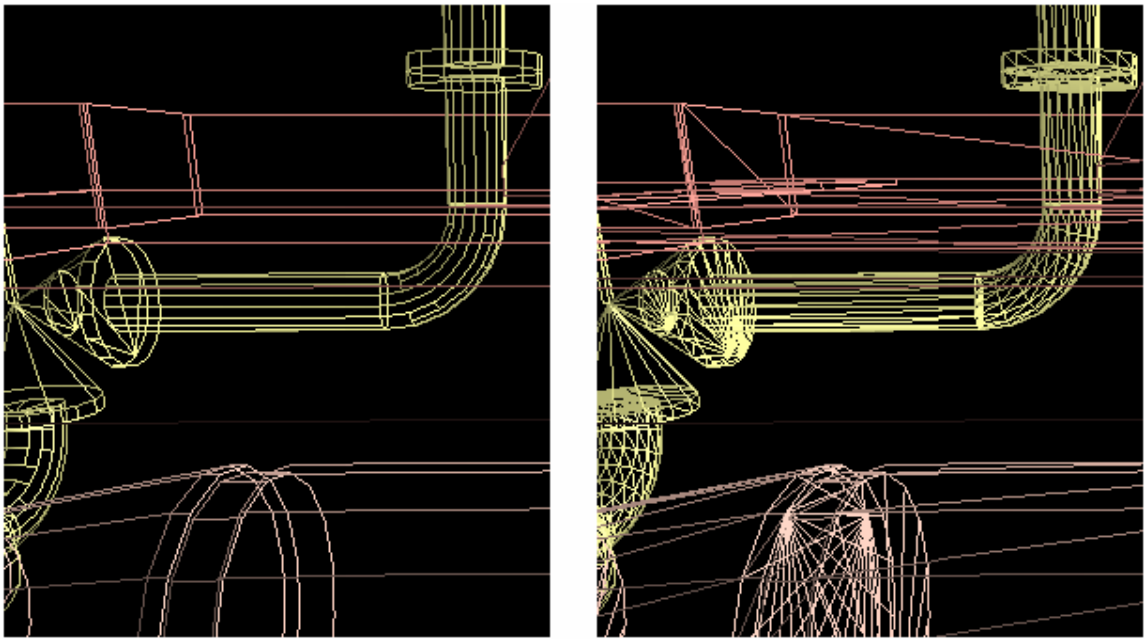
L'aspecte dels gràfics obtinguts és molt similar al de la figura 8.3.2.1 (comparació dels tipus de dades utilitzats): el rendiment és molt similar en tota la navegació, excepte en l'interval entre els frames 800 i 1.200, on la supressió de les coordenades de textura és una mica millor. Aquest fet dona encara més motius per pensar que la discrepàncies en aquesta fase de la navegació (en els dos casos), són provocades per un factor aliè a la variable que s'està analitzant. Així doncs, es considera que les variacions en l'eficiència són despreciables. Per tant, es desestima fer un tractament d'aquest tipus per als models no texturats.

8.3.5 Processament de la geometria

El desenvolupament d'aquesta optimització aplica una sèrie de transformacions sobre la geometria del model original. Així es permet adaptar-la convenientment a les necessitats dels *VBOs*. D'aquesta manera, esdevé necessari assegurar-se que aquestes conversions s'efectuen correctament i que no apareix cap tipus

d'imprevist al respecte. Per aconseguir-ho, s'analitza quina és la geometria de l'escena en les diferents etapes d'aquest procés de conversió.

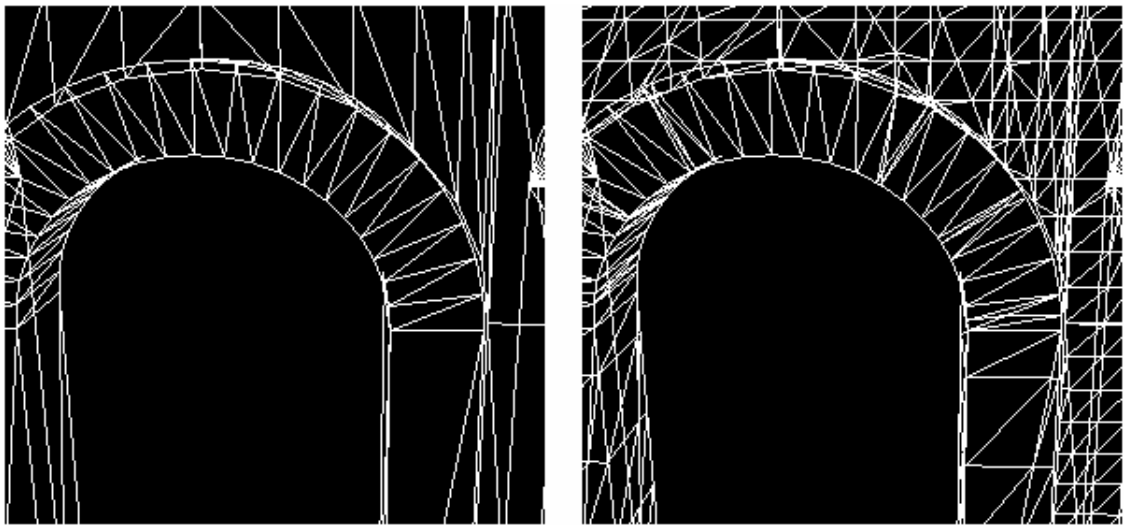
En una primera fase, es genera una malla de triangles per cadascun dels objectes del model. Inicialment, aquests objectes es representen mitjançant una llista de primitives *OpenGL*. Per tant, resulta imprescindible triangular-les per a poder construir una malla. A les imatges de la figura 8.5.3.1, es pot veure quin aspecte té la geometria abans i després de triangular-ne les primitives. Sembla que el càlcul està funcionant correctament ja que no s'hi detecta cap anomalia.



(a) *(b)*

Fig. 8.3.5.1: *Resultat de triangular les primitives del model per a poder usar VBOs.*
(a) Malla original (b) Malla triangulada

Tot seguit, en els models que utilitzen textures, cal fer un tractament especial als triangles per a què funcionin correctament els modes d'adreçament. Tal i com explica el punt 6.3.1.4 d'aquesta memòria, s'han de subdividir els triangles que tinguin coordenades de textura fora del rang [0,1]. Això permet fer *wrapping* d'una textura continguda dins l'atlas. A la figura 8.3.5.2, es mostra quin és l'aspecte de la malla un cop realitzada aquesta subdivisió. S'observa amb facilitat que apareix una quadricula sobre els triangles de la paret. Cada cel·la d'aquesta quadrícula utilitza el contingut de tota la textura que té definida la paret. Així es permet suportar la repetició mantenint sempre les coordenades de textura entre 0 i 1.



(a) *(b)*

Fig. 8.3.5.2: *Subdivisió de la geometria del model segons les coordenades de textura.*
(a) Model original (b) Geometria subdividida

Desafortunadament, a la imatge que s'obté un cop es fa el *rendering* de l'escena s'hi observen *artifacts* (veure figura 8.3.5.3). Es tracta d'unes línies de diferent color al de la textura usada, que apareixen a la frontera dels polígons que defineixen la quadrícula generada. S'intueix que això té a veure amb algun tipus de filtrat que fa *OpenGL* (ex. *antialiasing de les arestes dels polígons*), però no es té la seguretat que aquest en sigui el motiu. En qualsevol cas, desactivar aquest tipus de filtres no n'és la solució, ja que llavors s'haurien de resoldre nous problemes.

(a) *(b)*

Fig. 8.3.5.3: *Artifacts que apareixen a les vores de les textures que utilitzen els modes d'adreçament.*
(a) Model original (b) Geometria subdividida

Una forma alternativa de donar suport als mode d'adreçament en un *atlas* de textura seria fent servir un *pixel shader* [25]. Aquest és capaç d'emular el *wrapping* a partir de redireccionar adequadament l'accés a textura que fa cada fragment. D'aquesta manera, no faria falta fer la subdivisió de la geometria segons les coordenades de textura. Com a conseqüència directa, s'eliminarien els *artifacts* anteriors i la complexitat del model no es veuria afectada.

Malauradament, no s'ha tingut coneixement d'aquesta tècnica fins a la fase final d'aquest projecte, cosa que ha suposat que no poder disposar de temps suficient per a implementar-la. Per tant, es deixa com a tasca pendent per a treballs futurs relacionats amb aquesta tesi.

8.4 Impostors

Els experiments relatius a l'ús d'impostors tenen com a única finalitat el càlcul del valor òptim per al llindar d'impostors. Aquest, indica a partir de quin punt, els impostors generats perden la seva validesa i no poden ser utilitzats per al *rendering* (veure el punt 6.4 per a una descripció detallada). Així doncs, de forma similar al que passava amb el llindar de visibilitat, aquí també cal trobar un valor que estabilitzi el guany aconseguit en relació amb la qualitat de les imatges obtingudes.

Els tests realitzats prenen diferents valors per al llindar d'impostors, de manera que sigui possible comparar-los. D'aquesta manera, a la gràfica de la figura 8.4.1, es pot veure visualment quin és el rendiment de cadascun dels valors:

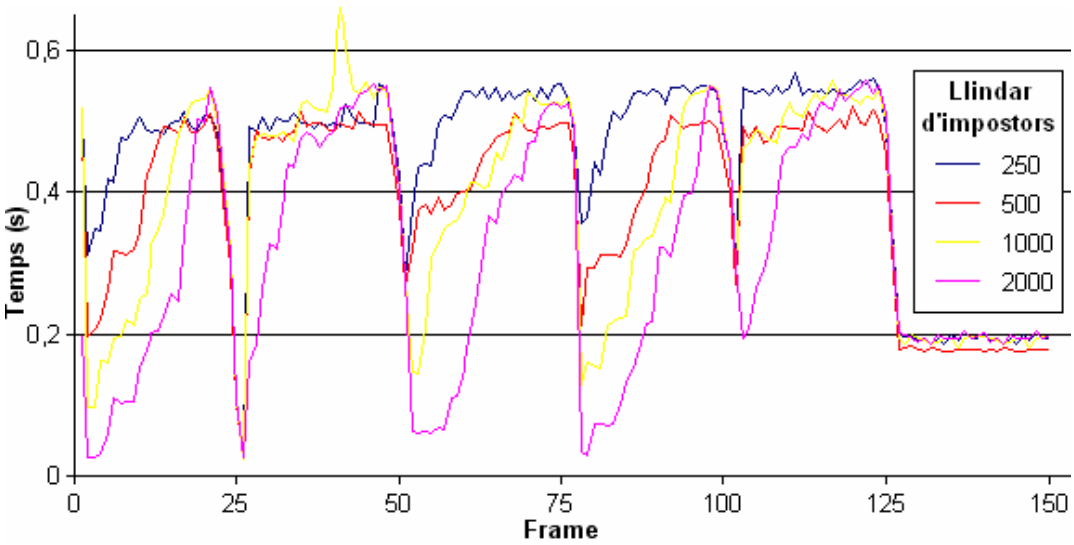


Fig. 8.4.1: *Gràfica comparativa utilitzant diferents valors per al llindar d'impostors.*

D'entrada, cal comentar la forma de les corbes obtingudes. En tots els casos, la navegació es pot descomposar en cinc intervals on el temps de generació d'un *frame* va augmentant de forma progressiva fins a un màxim i després descendeix sobtadament. Precisament, cadascuna d'aquestes fases encaixa perfectament amb una de les aproximacions que conformen el recorregut realitzat. Això es justifica sabent que els impostors redueixen molt el cost del procés de visualització quan la geometria es troba lluny de l'observador, però que han de deixar d'utilitzar-se a mesura que aquest si acosta (per evitar que es produeixin *artifacts*).

Passant a analitzar les diferències entre execucions, es pot veure amb claredat que el temps de *frame* s'incrementa a mesura que el valor del llindar d'impostors disminueix. Aquest fet succeeix degut a que la vida dels impostors s'escurça quan el llindar és més baix. També cal accedir als impostors de nivells més profunds de l'estructura jeràrquica, cosa que té un major cost. Tot i això, en els instants en que el punt de vista es troba molt a prop del centre model, el rendiment passa a ser molt similar. Això és així perquè no resulta possible utilitzar cap dels impostors generats en cap dels tests.

Respecte a la qualitat, aquesta pot contrastar-se mitjançant les imatges de les figures 8.4.2 i 8.4.3. En aquestes, es mostra una vista de l'escena, amb la mateixa orientació de la camera, però situant l'observador a diferents distàncies. En particular, les captures de la figura 8.4.3 estan agafades des del punt més pròxim al model on es permet l'ús d'impostors, segons els diferents valors del llindar. Es tracta de les situacions on els impostors difereixen més de la imatge real. La figura 8.4.2 en canvi, mostra una imatge de la geometria original del model (sense fer ús d'impostors) per a propòsits de comparació.

El test on el llindar té un valor de 250 pixels, gairebé no activa mai el mecanisme d'impostors. De fet, per a poder fer-ne ús, cal situar l'observador excessivament lluny, de manera que el model estigui més enllà del pla *z-far* del *frustum* de visió. Quan el llindar val 500 o 1.000, els impostors entren en joc i com a conseqüència, les imatges produïdes comencen a presentar alguns *artifacts*, tot i que tolerables (degut al reduït tamany dels nodes). Finalment, en l'execució amb valor 2.000, l'error introduït ja és massa gran, fent que les imatges produïdes no puguin considerar-se com a vàlides.

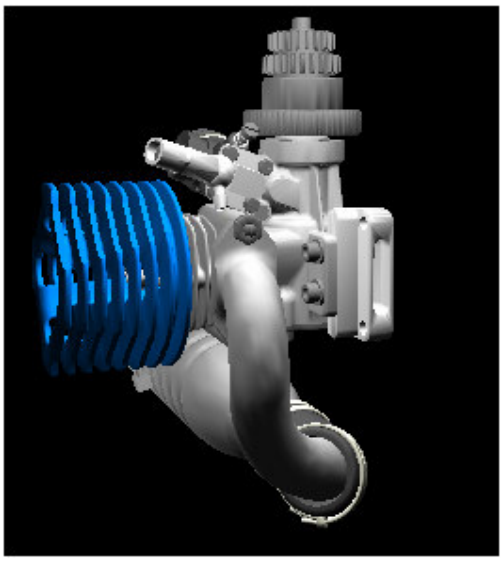
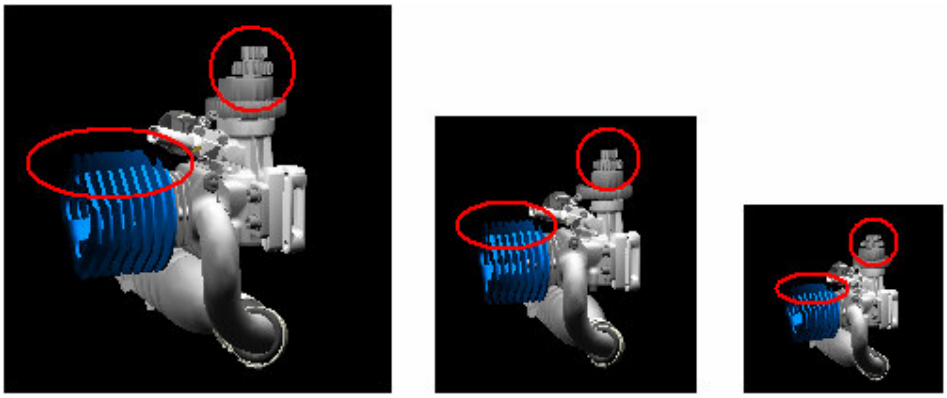


Fig. 8.4.2: *Screenshot del model Motorcillo.P3D renderitzat sense utilitzar impostors.*



(a) *(b)* *(c)*

Fig. 8.4.3: *Imatge generada fent ús d'impostors. Es pren la vista més gran que possible que els fa servir per cada valor del llindar. (a) Valor 2.000 (b) Valor 1.000 (c) Valor 500*

Així doncs, s'agafa 1.000 com a valor definitiu per al llindar d'impostors. Es tracta del nombre de pixels que major rendiment obté i que a la vegada no produeix *artifacts* inacceptables.

En aquest moment, ja s'està en disposició d'escollir quina resolució hauran de tenir les textures dels impostors. Tal i com es comenta prèviament, aquestes haurien de tenir un tamany de l'ordre del de la projecció màxima d'un node en pantalla. Suposant que la projecció i les textures siguin quadrades, amb un llindar de 1.000 pixels, es requereix com a mínim una textura de 32 x 32 ($\sqrt{1000}$ = 31,623 ≈ 32). Com que es vol que les textures a suportin correctament les

deformacions de projecció, se'ls hi dona una resolució superior a aquest mínim. A més, donat que es té la restricció de que les dimensions siguin múltiples de 2, l'increment es fa fins a un tamany de 64 x 64.

En últim terme, la mida definitiva de les textures es defineix en funció de la relació d'aspecte de les cares de les regions del *K-d tree*. Com que s'ha de satisfer la restricció de múltiples de 2, s'utilitza una taula que indica quina és el tamany més apropiat per cada rang de valors de l'*aspect ratio*. És la següent:

Aspect Ratio	Resolució textura
0 – 0,375	16 x 64
0,375 – 0,75	32 x 64
0,75 – 1,5	64 x 64
1,5 – 2,5	64 x 32
2,5 - ∞	64 x 16

Fig. 8.4.4: *Dimensions de la textura d'un impostor en funció de la relació d'aspecte de la cara corresponent d'un node del K-d tree.*

8.5 Tests generals

Les proves realitzades en aquest apartat no es focalitzen en cap optimització concreta, sinó que pretenen donar una visió global dels canvis realitzats. En primer lloc, es vol fer una comparativa del guany aportat per cadascuna de les millores implementades respecte l'aplicació original. Addicionalment, també es vol quantificar com incrementa el temps de càrrega dels models, l'espai desaprofitat per l'atlas de textura i l'augment de complexitat del model.

8.5.1 Comparativa de rendiment

Per tal d'avaluar quins desenvolupaments produeixen un major benefici, s'han efectuat una sèrie d'execucions activant i desactivant les noves tècniques d'acceleració. En totes les proves realitzades, s'assoleixen uns resultats de característiques molt similars. D'aquesta manera, només es mostren els resultats d'una de les execucions realitzades. Són els següents:

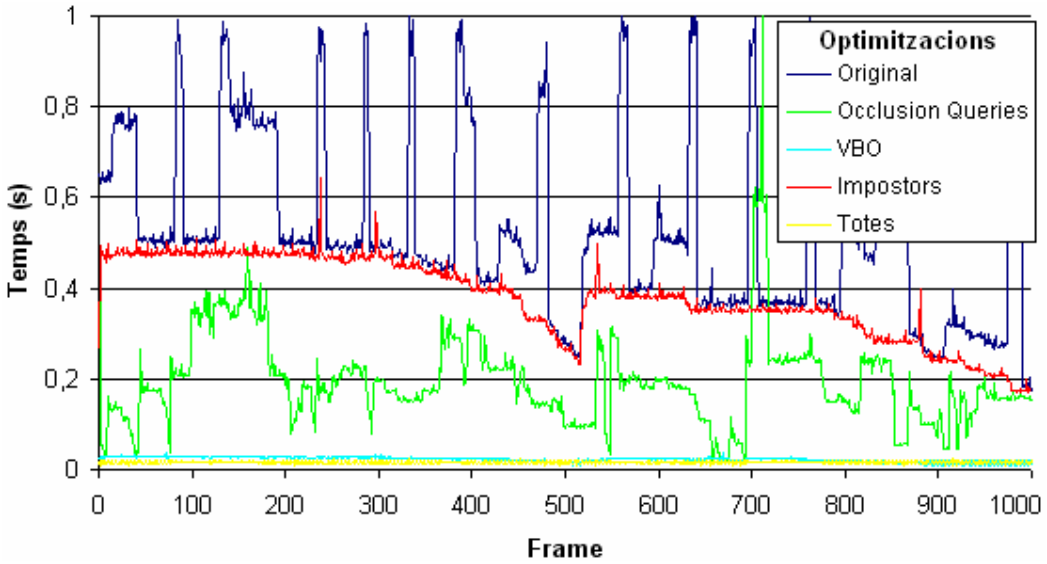


Fig. 8.5.1.1: *Gràfica comparativa del rendiment de les optimitzacions implementades. Execucions realitzades en el model Texaco.P3D (replicat 5 vegades).*

Els resultats obtinguts mostren amb contundència que el rendiment de l'aplicació s'incrementa de forma destacable. Es passa d'un *framerate* de 1,93 fps sobre l'aplicació original a una velocitat de 62,75 fps quan s'activen totes les propostes implementades. Així doncs, queda perfectament justificat que tot el treball realitzat permet millorar l'aplicació.

Tot i això, cal remarcar que la major part d'aquest benefici prové de l'optimització relativa als *VBOs*. Si es posa èmfasi en l'execució on només hi ha activa aquesta implementació, es pot veure que el guany obtingut és extremadament gran. De fet, amb només els *VBOs* el temps de *frame* mitjà passa de 0,518 s a 0,022 s, mentre que en l'aplicació final és de 0,016 s. Això demostra que el coll d'ampolla d'*Alice* realment es troba en la transferència de dades a la *GPU*, tal i com s'havia dit en l'anàlisi d'alternatives.

El segon desenvolupament amb més benefici és el que fa referència a les *hardware occlusion queries*. Tot i no ser tan espectaculars com els *VBOs*, són capaces de gairebé triplicar la velocitat del navegador (obtenen un *framerate* mitjà de 5,14 fps). En qualsevol cas, s'ha de tenir present que el seu comportament està subjecte a les característiques de l'escena i la navegació utilitzades (grau d'oclusió). Concretament, en les altres proves realitzades, els resultats obtinguts han estat de menor abast.

En última instància, ens trobem amb la substitució de la geometria llunyana per polígons texturats (impostors). L'aportació d'aquesta tècnica és insignificant en relació amb la dels altres desenvolupaments (només puja fins a 2,63 fps). Per tant, encara que fos la optimització de la qual se n'esperés menys, els resultat obtinguts són decepcionants. De totes maneres, el comportament dels impostors també es veu molt influenciat pel tipus de navegació realitzada i probablement, la que s'ha seleccionat no n'explota totes les possibilitats. A més a més, la utilització de tipus d'impostors més complexes, com ara els *relief impostors* [17] o els *ORI's* [18], permetria allargar-ne la vida i com a conseqüència, fer més eficient tot el procés de visualització.

Finalment, només resta comentar que l'impacte de les modificacions realitzades és més important com major és la complexitat del model. En les proves que s'han efectuat, es detecta que un increment important en el volum de geometria penalitza molt poc el *framerate*. Per exemple, mentre en un model de 250.000 triangles s'obtenen 63,99 fps, en un de 5.000.000 de triangles se n'aconsegueixen 62,75 fps. Aquest efecte, provocat pels *VBOs*, fa que la major part del temps de *frame* es consumeixi amb tasques que no depenen de la quantitat de geometria del model. Així es permetrà que *Alice* treballi amb escenes molt grans, mantenint les prestacions de temps real requerides.

8.5.2 Temps de càrrega

Donat que s'han introduït nous preprocessos de cost no despreciable, no hi ha cap mena de dubte que el temps necessari per a inicialitzar l'escena seleccionada es veurà incrementat. Per determinar quin és l'abast d'aquest augment, es mesura, per a diversos models, quin és el temps requerit per a llegir-lo en l'aplicació original i en la versió modificada d'aquesta. Els resultats obtinguts es presenten a la gràfica de la figura 8.5.2.1.

En tots els experiments realitzats, es pot veure amb claredat que el codi modificat multiplica el temps de càrrega de la revisió de partida. En particular, s'observa que en els models de major complexitat (en els que es fan rèpliques de la geometria), la diferència s'amplia encara més. Això es justifica en base a que els càlculs que efectuen el processament de la informació geomètrica no tenen un cost lineal.

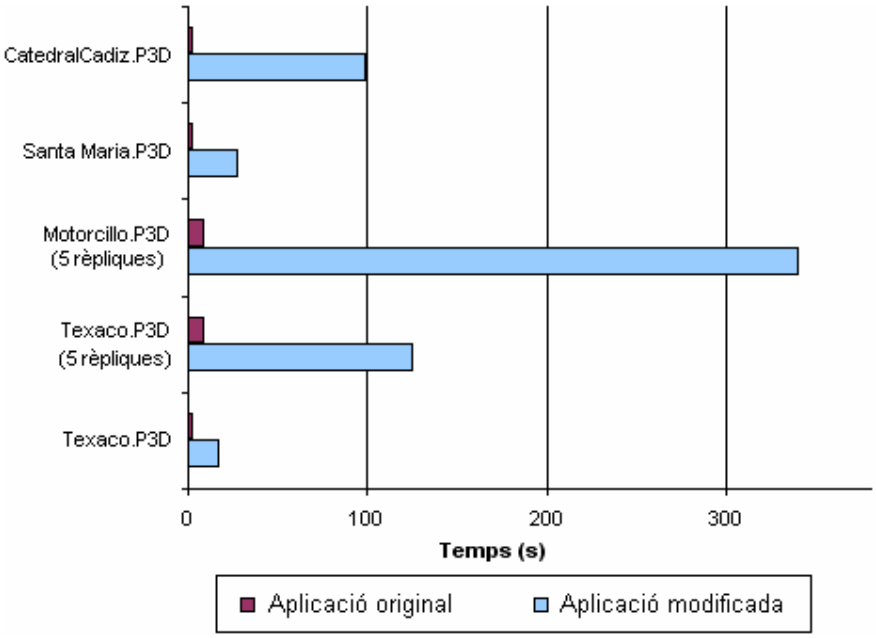


Fig. 8.5.2.1: *Temps de càrrega dels models a l'aplicació modificada en relació amb el temps de l'aplicació original.*

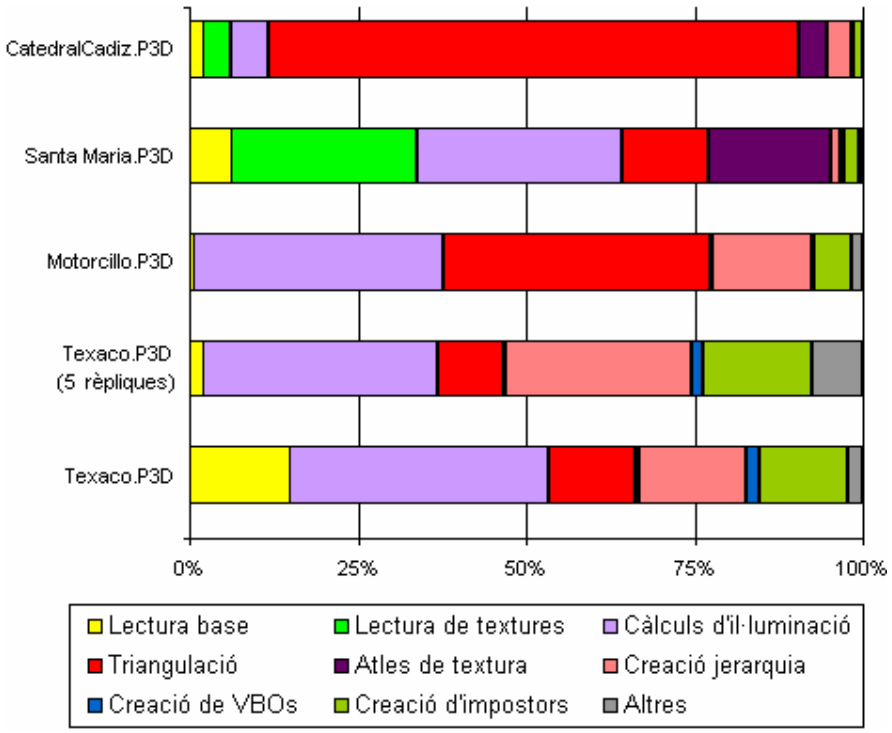


Fig. 8.5.2.2: *Percentatge del temps de càrrega de cadascun dels preprocessos executats.*

Per altra banda, també es té interès en veure quina proporció del temps de lectura ocupa cadascun dels preprocessos que s'executen. Així doncs, a la figura 8.5.2.2 es desglossa el cost total de carregar els models utilitzats en cadascuna de les fases que es porten a terme.

D'entrada, es pot veure que les operacions que resulten més cares en tots els casos són els càlculs d'il·luminació i la generació de la malla de triangles. Els segueixen de prop la construcció de l'estructura jeràrquica i la creació d'impostors. Tot i això, en els models texturats, la lectura de les textures i la creació de l'atlas també acostumen a tenir un pes important. Finalment, la creació dels *VBOs* no representa un cost significatiu respecta al total.

A més, en les escenes amb textures, el percentatge de temps necessari per a crear la malla és encara més gran. Això es justifica amb el tractament addicional que es fa per a suportar els modes d'adreçament. De fet, en el cas de la catedral de *Cádiz*, aquesta situació es porta a l'extrem degut a que el terra i les parets usen extensivament el *wrapping* de textures.

Abans d'acabar, cal recordar que la majoria d'aquestes fases, a excepció de la lectura base i la lectura de textures, es poden suprimir de la lectura modificant el format dels fitxers *P3D* (proposta del punt 4.2.1). Amb això, s'aconseguiria que l'usuari només hagués d'esperar un temps similar al de l'aplicació original abans de començar a interactuar amb el model.

8.5.3 Dimensions de l'atlas de textura

La optimització relativa als *VBOs* fa que sigui necessari utilitzar un atlas que condensi totes les textures del model en una de sola. Com que les textures *OpenGL* han de ser forçosament rectangulars, la superfície de l'atlas també tindrà aquestes característiques. D'aquesta manera, s'hi disposen totes les textures del model de manera que l'espai desaprofitat sigui mínim. Tanmateix, l'algoritme implementat per a efectuar aquesta distribució no és òptim (ha estat seleccionat per la seva simplicitat). Així doncs, l'objectiu d'aquest punt és valorar el comportament d'aquesta solució, cosa que es farà a partir de mesurar la memòria de textura que resta inservible.

En les dues escenes texturades (*CatedralCadiz.P3D* i *SantaMaria.P3D*), s'ha mesurat quin espai ocupa la suma de totes les textures i quines són les dimensions de l'atlas generat. A la taula següent, es mostren quines són les dades obtingudes:

Model	*CatedralCadiz.P3D*	*SantaMaria.P3D*
Núm. textures	119	149
Pixels suma textures	17.780.800	28.573.696
Pixels atlas	17.825.792	28.639.232
Percentatge desaprofitat	0,252 %	0,229 %

Fig. 8.5.3.1: *Taula que mesura la memòria de textura desaprofitada per l'atlas.*

Dels resultats anteriors, se'n desprèn que la quantitat de memòria de textura perduda en la construcció de l'atlas és mínima (inferior a un 1%). Això demostra que l'algoritme utilitzat, tot i no estar optimitzat, dona resultats molt vàlids. En qualsevol cas, caldria portar a terme més execucions sobre un rang més ampli de models per poder tenir la certesa de que aquest mètode no falla.

8.5.4 Increment de la complexitat

El tractament que s'aplica sobre la geometria del model fa que aquesta n'incrementi la seva complexitat. Per una banda, la triangulació de les primitives fa que se n'augmenti el nombre. Per l'altra, en les escenes texturades, cal subdividir els triangles per tal de que les coordenades de textura es mantinguin dins el rang [0,1]. D'aquesta manera, és una qüestió important conèixer com augmenta la complexitat dels models degut a aquests tractaments. Així doncs, utilitzant els resultats de les execucions anteriors, es construeix la gràfica de la figura 8.5.4.1.

D'entrada, es pot veure que la transformació de les llistes de primitives en malles de triangles fa que el nombre de triangles ascendeixi considerablement. En alguns casos, com ara en el model *Texaco.P3D*, l'increment és de més d'un 600%. Tot i això, s'ha de tenir present que es fa referència a primitives *OpenGL*. Això significa que construccions com ara *strips* o *fans* es comptabilitzen com a una única primitiva. Addicionalment, també s'ha de tenir en compte que el *rendering* de triangles és molt més eficient que el de qualsevol altra primitiva. Per tant, encara que les xifres obtingudes puguin semblar una altra cosa, la complexitat real pràcticament no es veu afectada.

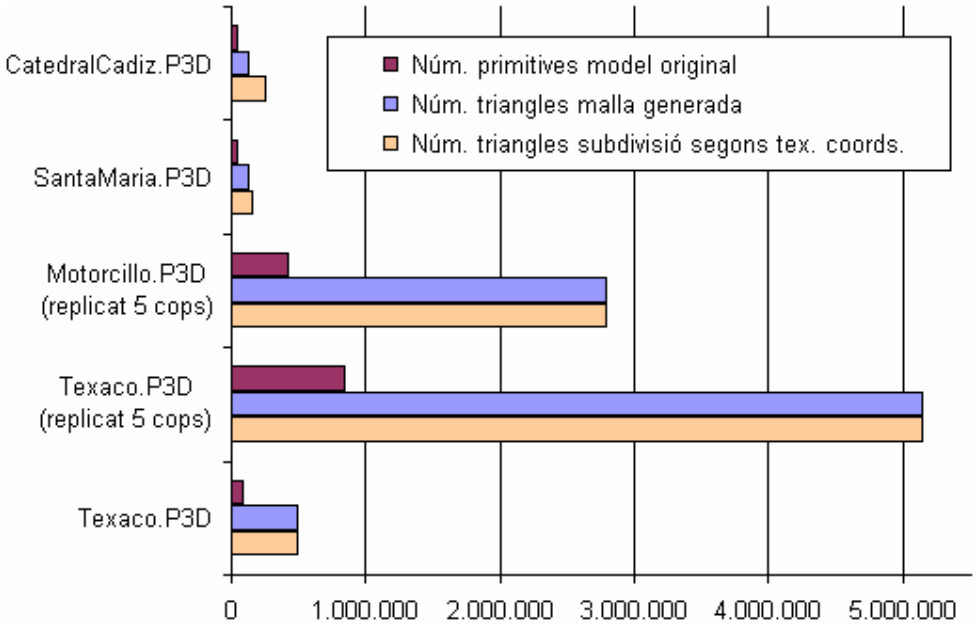


Fig. 8.5.4.1: *Increment de complexitat al convertir els models en malles de triangles.*

Pel que fa a la subdivisió segons les coordenades de textura, els valors obtinguts són molt variables. Mentre en l'escena de la catedral de *Càdiz* gairebé es duplica el nombre de triangles, en la de l'església de Santa Maria la diferència és molt menor. Això és així perquè l'efecte d'aquest processament varia en funció del grau d'utilització del mode d'adreçament per *wrapping* al texturar el model.

9 TREBALL FUTUR

Durant les diferents etapes del transcurs d'aquesta tesi, s'han anat trobat situacions en les què s'han deixat aspectes oberts. No s'ha disposat del temps suficient per estudiar tots aquests casos amb profunditat i s'ha preferit deixar-los com a punts de partida de nous treballs que parteixin dels resultats aconseguits en aquest. Cal pensar que el projecte *BAIP 2020* no finalitza fins a finals de l'any 2010 i per tant, encara hi ha marge per a fer moltes d'aquestes coses. A continuació, es resumeixen les principals possibles ampliacions que s'han pensat.

9.1 Optimitzacions no seleccionades

A l'anàlisi d'alternatives del punt 4, es presenta una llarga llista de possibles millores a realitzar. D'aquestes, només se n'ha implementat un petit subconjunt per a poder adequar-se al temps disponible. Així doncs, hi ha un nombre important de propostes en les quals no s'ha pogut treballar. En destaquen el canvi de format dels fitxers *P3D* (permeten carregar els models més ràpidament), l'ús de tècniques out-of-core (suporten models molt més complexes, fins i tot gegantins) o la paral·lelització de l'execució del codi mitjançant *multithreading* (fan ús dels múltiples nuclis dels computadors actuals). Per tant, el desenvolupament de totes les optimitzacions descartades esdevé una clar camí per a continuar millorant *Alice*.

9.2 Wrapping de textures mitjançant un fragment shader

La proposta relativa als *Vertex Buffer Objects* (*VBOs*) obliga a fer modificacions importants per a gestionar correctament els models texturats. Concretament, cal construir un *atlas* de textura per a poder *renderitzar* la geometria en un únic pas, sense haver de canviar l'estat d'*OpenGL*. Això, a la vegada, té repercussions sobre el comportament dels modes d'adreçament de les textures (el tractament que es dona a les coordenades de textura fora l'interval [0,1]). Per tal que aquests modes funcionin bé, es subdivideixen els triangles de l'escena, assegurant que les coordenades de textura no surtin del rang [0,1]. Tot i això, en les proves realitzades es pot veure que aquesta tècnica prova *artifacts* importants.

Una solució de la qual s'esperen millors resultats és la proposada a [25]. Es tracta d'implementar el *wrapping* de textures mitjançant un *fragment shader*. Així, es pot gestionar amb facilitat quin o quins accessos a textura ha de fer cada fragment per il·luminar el pixel corresponent. D'aquesta manera, no cal subdividir la geometria i com a conseqüència, s'eviten els *artifacts* anteriors. Malauradament, aquesta tècnica no s'ha conegut fins a la fase final del projecte i no s'ha disposat de temps suficient per a implementar-la. Com que la idea que planteja és molt prometedora, caldrà tenir-la molt present quan es decideixi continuar amb el treball iniciat aquí.

9.3 Impostors més sofisticats

L'ús d'impostors per a la representació de la geometria llunyana és la tècnica d'acceleració implementada que pitjors resultats dona. Això es deu a la curta vida que han de tenir els polígons texturats per a poder evitar que es produeixin *artifacts* notables. Aquests només són vàlids des d'un reduït conjunt de punts de vista, ja que no suporten efectes de *parallax*.

Per altra banda, tal i com es comenta en el punt 4.4.2, existeixen altres tipus d'impostors, generalment més sofisticats que els simples polígons texturats. Es tracta dels *relief impostors* [17] o dels *ORIs* [18]. Aquests, tot i ser una mica més complexes, poden ser utilitzats des d'un major nombre de posicions i orientacions de l'observador. Amb aquesta filosofia, s'haurien d'assolir fites de rendiment molt més elevades. Per tant, una altra possible ampliació d'aquesta tesi consistiria en implementar aquests altres tipus impostors.

9.4 Ús de la il·luminació d'OpenGL

Actualment, *Alice* computa la il·luminació de cada vèrtex per software. Així doncs, en un procés que s'executa durant la càrrega del model, es calcula quina influència provoquen les llums de l'escena en el color dels vèrtexs. D'aquesta manera, s'estan ignorant per complet els càlculs d'il·luminació que ofereix *OpenGL*. Això es feia així perquè, històricament, el hardware gràfic penalitzava molt aquest tipus d'operacions. Avui dia però, les targetes gràfiques modernes ho suporten perfectament i per tant, aquest preprocés ja no té raó de ser. A més, tampoc s'està

considerant la component especular de la il·luminació, ja que aquesta depèn de la posició de l'observador (no pot calcular-se a priori).

Amb aquest plantejament, l'actualització de la il·luminació de l'aplicació es fa molt necessària. Tanmateix, el canvi proposat no és trivial. El sistema actual permet definir llums de zona, de manera que només afectin a uns objectes determinats. Això obliga a modificar l'estat de les llums per cada objecte que es renderitza, cosa que xoca frontalment amb l'ús dels *VBOs* (interessa que l'estat d'*OpenGL* es mantingui constant). Per tant, és evident que fa falta treballar més en aquest punt per a resoldre aquests problemes.

9.5 Restauració de funcionalitats perdudes

Les primeres versions d'*Alice* disposaven d'algunes funcionalitats addicionals, tal i com es pot veure en el manual de l'aplicació [30]. Es permetien fer coses com ara treballar interactivament amb trajectòries, manipular la geometria de l'escena, texturar polígons de forma interactiva, ... Tot i això, degut a algunes de les evolucions que ha sofert el navegador, totes aquestes característiques s'han perdut. Algunes de les comandes relacionades s'han deshabilitat i les altres no acaben de funcionar del tot bé. Per aquest motiu, es suggereix restaurar les funcionalitats perdudes i així dotar a l'aplicació de les capacitats funcionals que tenia inicialment.

10 ESTIMACIO INICIAL I ESFORÇ REAL

Un cop s'han acabat totes les tasques que formen part d'aquesta tesi de màster, es pot determinar el grau en què s'ha complert la planificació inicial. A la taula 10.1 es mostra, per a cadascuna de les tasques contemplades, la relació entre les hores previstes i les que s'hi han dedicat en realitat.

En els resultats globals de la taula es pot veure que la planificació realitzada s'ha complert. El nombre total d'hores realitzades (590) es troba comprès en el rang de valors que es va calcular (entre 540 i 594). Això era d'esperar donat que el nombre d'hores disponibles és una de les limitacions del projecte. Per això, per poder tenir una idea més precisa de com ha evolucionat el desenvolupament, cal analitzar amb detall cadascuna de les seccions i en els punts on hi hagi les principals discrepàncies, buscar-ne les causes.

Respecte a l'anàlisi d'alternatives, aquest s'ha pogut completar consumint el nombre d'hores que tenia assignades. Tot i això, les primeres temàtiques que s'han estudiat s'han considerat molt interessants i s'ha preferit treballar-les amb més profunditat. De fet, els desenvolupaments que s'han acabat escollint fan referència a aquests àmbits. La qüestió és que s'hi ha invertit més temps del previst i això ha provocat que els darrers punts s'hagin tractat de forma més superficial.

Pel que fa a la fase de desenvolupament tècnic, tal i com es pot observar, la optimització dels *VBOs* és el que difereix més del previst (s'ha requerit gairebé el doble). Si es mira amb detall, es pot veure que aquesta desviació prové principalment de la creació de l'atlas i de la subdivisió de la geometria per a suportar el *wrapping* de textures. Es tracta de dues tasques que no entraven dins la planificació inicial, entre altres coses, perquè no s'havia considerat que la gestió de textures en els *VBOs* fos tan complexa.

Afortunadament, la resta de millores seleccionades s'han ajustat bastant bé a l'estimació temporal. En algun cas, com ara en les *hardware occlusion queries*, fins i tot s'ha pogut completar la implementació més aviat del compte. Gràcies això i a les hores reservades per a contingència de riscs, l'error de previsió dels *VBOs* s'ha pogut compensar. Tanmateix, també hauria resultat interessant disposar de més temps per poder afinar més bé el funcionament dels impostors.

Nom de la tasca	Duració estimada	Duració real
- Plantejament inicial	**60 h**	**50 h**
Definició del projecte	**10 h**	**10 h**
Selecció del visualitzador de partida	**20 h**	**20 h**
Preparació de l'entorn de treball i instal·lació de software	**30 h**	**25 h**
- Estudi de possibles millores per accelerar el rendiment d'*Alice*	**120 h**	**130 h**
GPUs i novetats d'OpenGL	**15 h**	**25 h**
Estructures de dades	**15 h**	**25 h**
Visibilitat	**15 h**	**20 h**
Multiresolució	**15 h**	**10 h**
Out-of-core	**15 h**	**10 h**
Multithreading	**15 h**	**10 h**
Documentació	**30 h**	**30 h**
- Desenvolupament tècnic de les optimitzacions seleccionades	**300 h**	**350 h**
Vertex Buffer Objects	**60 h**	**115 h**
Conversió de la geometria en malles de triangles	20 h	15 h
Operador de gestió de VBOs	10 h	10 h
Generació dels VBOs a partir de les malles de triangles	20 h	25 h
Modificació del procés de rendering per a utilitzar els nous VBOs	10 h	10 h
Creació de l'atlas de textura	---	25 h
Subdivisió segons coordenades de textura (wrapping)	---	30 h
Estructura jeràrquica	**45 h**	**35 h**
Estructura de dades	10 h	10 h
Algoritme de selecció del pla de divisió	20 h	15 h
Gestió dels objectes fragmentats pel pla de divisió	15 h	10 h
Occlusion Queries	**40 h**	**30 h**
Operador de gestió de les hardware occlusion queries	10 h	10 h
Algoritme de rendering basat en Coherent Hierarchical Culling	30 h	20 h
Impostors	**50 h**	**55 h**
Generació de les textures utilitzades en els polígons impostors	35 h	30 h
Algoritme que decideix quan s'han d'utilitzar els impostors	15 h	25 h
Contingència de riscs de la fase d'implementació	**15 h**	**---**
Disseny i execució d'experiments	**30 h**	**50 h**
Documentació	**60 h**	**55 h**
- Altres	**60 h**	**60 h**
Gestió del projecte (planificació, reunions, ...)	**15 h**	15 h
Confecció de la memòria	**30 h**	30 h
Presentació / lectura de la tesi	**15 h**	15 h
SUBTOTAL	**540 h**	
CONTINGENCIA DE RISCS (10 %)	**54 h**	**---**
TOTAL	**594 h**	**590 h**

Fig. 10.1: *Comparació de la planificació inicial amb l'esforç que realment s'ha realitzat.*

Passant a l'etapa de disseny i execució d'experiments, cal comentar que també s'ha excedit el temps disponible. En aquesta ocasió, les discrepàncies s'originen en la falta de models de gran tamany per efectuar proves d'estrès. Per resoldre aquesta mancança, s'ha implementat un mecanisme de generació de rèpliques (ha requerit hores addicionals). A més, tot i que aquest sistema permet realitzar les proves, no es tracta d'una solució òptima ja que complica el procés d'obtenció de resultats.

Finalment, només queda dir que les altres tasques previstes (definició del projecte, memòria, gestió, ...) s'han portat a terme sense alterar excessivament la planificació.

11 CONCLUSIONS

Finalment es recullen les conclusions que s'extreuen del desenvolupament de tota el treball realitzat. Es fa un balanç dels objectius assolits, es comenta l'aportació que es creu que pot tenir aquesta tesi en el projecte BAIP 2020 i per últim, es fa una petita reflexió personal.

11.1 Objectius assolits

Una vegada completada aquesta tesi, es pot afirmar que s'han assolit, pràcticament en la seva totalitat, els dos grans objectius marcats a l'inici del projecte. Per tant, els resultats obtinguts són els que s'esperaven i es pot dir que el projecte ha estat tot un èxit.

El primer d'aquests objectius consisteix en fer un anàlisi de les possibles millores per a accelerar el rendiment del procés de visualització d'*Alice*. En particular, es volia treballar en l'explotació de les característiques de les plaques gràfiques modernes, així com els múltiples *cores* dels computadors actuals. Així doncs, s'ha fet un estudi que recull una extensa llista de propostes, classificades segons la temàtica a què fan referència (veure el punt 4 d'aquesta memòria). A més, per cadascuna de les optimitzacions plantejades, s'indiquen les particularitats de la seva integració a *Alice* i el benefici que s'espera que proporcionin. Per tant, és evident que l'objectiu marcat s'ha assolit de manera satisfactòria.

L'altre objectiu plantejat tracta sobre la implementació d'una selecció de les optimitzacions que s'han suggerit. D'aquesta manera, s'han escollit quatre propostes de l'anàlisi anterior (en funció del guany previst, el temps disponible i les relacions de dependència entre alternatives): la implantació d'una estructura jeràrquica de divisió de l'espai, l'occlusion culling mitjançant *hardware occlusion queries*, el *rendering* eficient de la geometria a partir de *Vertex Buffer Objects* i la representació de la geometria llunyana fent ús d'impostors.

Addicionalment, s'han realitzat una sèrie d'execucions que permeten quantificar el guany obtingut amb cadascun dels canvis introduïts. En general, s'ha pogut veure que les propostes implementades milloren espectacularment el rendiment de l'aplicació original. Per fer-se'n una idea, només cal pensar que en algun cas on el

framerate del navegador ha passat de menys de 2 fps a gairebé 63 fps. Així doncs, es pot dir que aquest segon objectiu també s'ha complert.

Tot i això, en un dels desenvolupaments no s'ha obtingut el benefici que s'havia pronosticat. És el cas dels impostors, que pràcticament no tenen impacte en el rendiment de l'aplicació. Això es deu a l'excessiva simplicitat del tipus d'impostors utilitzats (polígons texturats), la qual fa que siguin vàlids (sense produir *artifacts*) en molt poques ocasions. Per millorar aquesta situació, es suggereix fer servir tipus impostors més sofisticats, de manera es suportin millor els efectes de *parallax* i com a conseqüència, aquests puguin entrar en joc més vegades.

Malgrat aquest punt dèbil i altres limitacions de caràcter menor, no es considera que siguin problemes irresolubles. De fet, en tots els casos s'han donat solucions per a resoldre'ls en futurs treballs. Per tant, tot i que els objectius particulars d'aquesta tesi s'han assolit, encara resten moltes coses pendents de fer.

11.2 Aportació d'aquesta tesi al projecte BAIP 2020

Tota la feina feta durant aquesta tesi s'ha desenvolupat en el context del projecte d'investigació *BAIP 2020*. Tal i com s'ha explicat prèviament, un dels grans objectius d'aquest projecte és la creació de software innovador per al disseny i la construcció de vaixells. En particular, una de les aplicacions que es volen obtenir és un visualitzador immersiu que suporti models de molt alta complexitat (ex. vaixell sencer) i a més, permeti navegar-hi en temps real.

Donat que es vol utilitzar *Alice* com a punt de partida d'aquest navegador, es fa imprescindible millorar-lo per a poder satisfer els requisits exigits. Així doncs, l'anàlisi d'alternatives fet en la primera part d'aquest treball fa un recull de propostes per a optimitzar el rendiment d'*Alice* i per a incrementar-ne la capacitat (tamany dels models suportats). Cal destacar que algunes d'aquestes propostes fan ús de les noves capacitats incloses en el *hardware* modern de gamma mitjana (principalment, *GPUs* i múltiples nuclis). Precisament, s'espera que aquestes siguin les opcions que produeixin un impacte més fort.

A més a més, la segona part del treball consisteix en la implementació de quatre dels canvis proposats. Com que els resultats obtinguts han estat favorables, es pot dir que la nova versió de l'aplicació es troba més a prop del que es necessita

per al *BAIP* 2020. Bàsicament, s'ha treballat per augmentar el rendiment, però sempre amb la limitació de models que tinguin cabuda en memòria principal (no s'ha treballat amb tècniques *out-of-core*).

En qualsevol cas, tot i que s'ha fet molta feina, els objectius del projecte *BAIP 2020* encara no s'han assolit. Així doncs, caldrà seguir treballant-hi fins que s'aconsegueixin les fites marcades. S'ha de pensar que el projecte finalitza a finals de l'any 2010 i per tant, la dedicació està garantida almenys fins aquest moment. D'aquesta manera, en el punt 9 d'aquesta memòria s'enumeren els treballs futurs que es consideren més interessants, entre els quals hi ha les propostes que s'han descartat. Es creu que el *BAIP 2020* hauria de seguir per aquestes línies per tal de satisfer els requeriments plantejats el més aviat possible.

11.3 Valoració personal

Com a autor d'aquest projecte, valoro tota la feina realitzada de forma molt positiva. Considero que he acumulat una gran experiència al llarg de tot el temps que ha durat el desenvolupament. Cal resaltar que la temàtica abordada no es focalitza en un àmbit concret, sinó que es tracten problemes molt diversos del món dels gràfics per computador. Així doncs, es podria dir que aquest projecte m'ha servit per posar una mica d'ordre en el garbell de coneixements adquirits durant els cursos del màster i per aprendre a com utilitzar-los en situacions reals.

M'agradaria concloure aquesta valoració dient que estic plenament satisfet dels resultats obtinguts. No obstant, s'han anat deixat nombrosos aspectes que cal resoldre. Degut a la forta implicació amb el projecte *BAIP 2020*, s'espera que això es faci pròximament. D'aquesta manera, queda justificat que tot l'esforç realitzat acabarà tenint una aplicació pràctica, cosa que em resulta molt gratificant.

12 BIBLIOGRAFIA

[1] WILLIAMS, I., HART, E.: *"Efficient rendering of geometric data using OpenGL VBOs in SPECviewperf"*, Standard Performance Evaluation Corporation - White Paper (2005), http://www.spec.org/gwpg/gpc.static/vbo_whitepaper.html

[2] NVIDIA Corporation: *"Using Vertex Buffer Objects (VBOs)"*, White paper (2003), http://developer.nvidia.com/object/using_VBOs.html

[3] RICCIO, C.: *"OpenGL Vertex Buffer Object"*, (2006), http://www.g-truc.net/article/vbo-en.pdf

[4] ASHBAUGH, B., BERETTA, B., BROWN, P., EVERITT, E., KESSENICH, J., LEECH, J., LICEA-KANE, D., LICHTENBELT, B., PODDAR, B., ROELL, T., ROMANICK, I., SANDMEL, J., SCHELTER, J.P., STAUFFER, J., TRIANTOS, N., VOGEL, D.: *"GL_ARB_vertex_buffer_object specification"*, OpenGL Extension Registry (2006), http://www.opengl.org/registry/specs/ARB/vertex_buffer_object.txt

[5] BAVOIL, L.: *"Rendering Huge Triangle Meshes with OpenGL"*, University of Utah (2005), http://www.sci.utah.edu/~bavoil/opengl/

[6] CEBOLLADA, V.: *"Visualizador estereoscópico multipantalla basado en clusters de PCs"*, PFC, Universitat Politècnica de Catalunya (2005)

[7] BRUNET, P., SANTISTEVE, F.J., CHIARABINI, L., PATOW, G.A., STAFFETTI, E., SURINYAC, J.: *"Estructuras Geométricas Jerárquicas para la Modelización de Escenas 3D"*, Universitat Politècnica de Catalunya (1999), http://www.lsi.upc.edu/~pere/PapersWeb/MasterV2/Report99_6.pdf

[8] COHEN-OR, D., CHRYSANTHOU, Y., SILVA C.T., DURAND, F.: "A Survey of Visibility for Walkthrough Applications", IEEE Transactions on Visualization and Computer Graphics (2003) 9 (3), 412-431, http://orion.lcg.ufrj.br/cg2/downloads/sombras/Visibility%20Survey.pdf

[9] BITTNER, J., WIMMER, M., PIRINGER, H., PURGATHOFER, W.: *"Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful"*, Computer Graphics Forum (Eurographics 2004), 23, 3 (2004), 615-624, http://www.cg.tuwien.ac.at/research/vr/vhcull

[10] GARLAND, M., HECKBERT. S.: *"Surface simplification using quadric error metrics"*, Proceedings of ACM SIGGRAPH (1997), 209-216, http://graphics.cs.uiuc.edu/~garland/papers/quadrics.pdf

[11] ANDUJAR, C., AYALA, D., BRUNET, P.: *"Topology Simplification through Discrete Models"*, ACM Transactions on Graphics (2002), 20 (6), 88-105, http://www.lsi.upc.es/~virtual/home/papers/Topology.pdf

[12] FUNKHOUSER, T.A., SEQUIN, C.H.: *"Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments"*, Proceedings of the 20th annual conference on Computer Graphics and interactive techniques (1993), 247 – 254, http://portal.acm.org/citation.cfm?id=166149

[13] ALIAGA, D.G., LASTRA, A.: *"Automatic Image Placement to Provide A Guaranteed Frame Rate"*, Proceedings of the 26th annual conference on Computer graphics and interactive techniques (1999), 307 – 316, http://portal.acm.org/citation.cfm?id=311574

[14] DÉCORET, X., DURAND, F., SILLION, F., DORSEY, J.: *"Billboard Clouds for Extreme Model Simplification"*, Proceedings of ACM SIGGRAPH (2003), 689 – 696, http://artis.imag.fr/Publications/2003/DDSD03/bc03.pdf

[15] SILLION, F., DRETTAKIS, G., BODELET, B.: *"Efficient Impostor Manipulation for Real-Time Visualization of Urban Scenery"*, Proceedings of Eurographics (1997), 207-218, http://artis.inrialpes.fr/Publications/1997/SDB97

[16] DÉCORET, X., SCHAUFLER, G., SILLION, F., DORSEY, J.: *"Multi-layered impostors for accelerated rendering"*, Proceedings of Eurographics (1999), 18 (3), http://groups.csail.mit.edu/graphics/pubs/EG99-Decoret.pdf

[17] POLICARPO, F., OLIVEIRA, M.: *"Relief Mapping of Non-Height-Field Surface Details"*, Proceedings of ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games (2006), 55-62, http://www.inf.ufrgs.br/~oliveira/pubs_files/Policarpo_Oliveira_RTM_multilayer_I3D2006.pdf

[18] ANDUJAR, C., BOO, J., BRUNET, P., FAIRÉN, M., NAVAZO, I., VÁZQUEZ, P., VINACUA, À.: *"Omni-directional Relief Impostors (ORIs)"*, Proceedings of Eurographics (2007), 26, 553-560 (8), http://www.lsi.upc.edu/~moving/papers/ORIs.pdf

[19] ISENBURG, M., GUMHOLD, S.: *"Out-of-Core Compression for Gigantic Polygon Meshes"*, Proceedings of ACM SIGGRAPH (2003), 935-942, http://www.cs.unc.edu/~isenburg/papers/ig-ooccgpm-03.pdf

[20] CIGNONI, P., MONTANI, C., ROCCHINI, C., SCOPIGNO, R.: *"External Memory Management and Simplification of Huge Meshes"*, IEEE Transactions On Visualization and Computer Graphics (2003), 9, 4, 525-537, http://ieeexplore.ieee.org/iel5/2945/28176/01260746.pdf

[21] GOBETTI, E., MARTON, F.: *"Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge 3D Models on Commodity Graphics Platforms"*, Proceedings of ACM SIGGRAPH (2005), 878-885, http://www.cs.unc.edu/~isenburg/ papers/ig-ooccgpm-03.pdf

[22] REINERS, D.: *"OpenSG"*, OpenSG Forum (2003), http://www.vrmedialab.dk/projects/cave2002/materials/CPW02_opensg.pdf

[23] WILLIAMS, A.: *"Boost Documentation - Chapter 17 - Thread"*, (2007), http://www.boost.org/doc/libs/1_35_0/doc/html/thread.html

[24] ENGELSCHALL, R. S.: *"GNU Pth – The GNU Portable Threads"*, (2006), http://www.gnu.org/software/pth/pth-manual.html

[25] NVIDIA Corporation: *"Improve Batching Using Texture Atlases"*, White Paper (2004), ftp://download.nvidia.com/developer/NVTextureSuite/Atlas_Tools/Texture_Atlas_Whitepaper.pdf

[26] GAREY, M.R., JOHNSON, D.S.: *"Computers and Intractability: A Guide to the Theory of NP-Completeness"*, W.H. Freeman & Co. (1982), http://portal.acm.org/citation.cfm?id=574848

[27] MARTELLO, S., TOTH, P.: *"Knapsack problems – Algorithms and Computer Implementations"*, John Wiley & Sons, (1990), http://www.or.deis.unibo.it/kp/KnapsackProblems.pdf

[28] GREEN, S.: *"The OpenGL Framebuffer Object Extension"*, Game Developers Conference (2005), http://download.nvidia.com/developer/presentacions/2005/GDC/OpenGL_Day/OpenGL_FrameBuffer_Object.pdf

[29] JULIANO, J., SANDMEL, J.: *"EXT_framebuffer_object"*, SGI Extensions Registry (2005), http://oss.sgi.com/projects/ogl-sample/registry/EXT/framebuffer_object.txt

[30] CEBOLLADA, V., TRUEBA, R., VALOR, D.: *"Manual de usuario del visualizador Alice"*, Universitat Politècnica de Catalunya (2003)